Graphology of an Exploit – Hunting for exploits by looking for the author's fingerprints

research.checkpoint.com/2020/graphology-of-an-exploit-volodya/

October 2, 2020

October 2, 2020

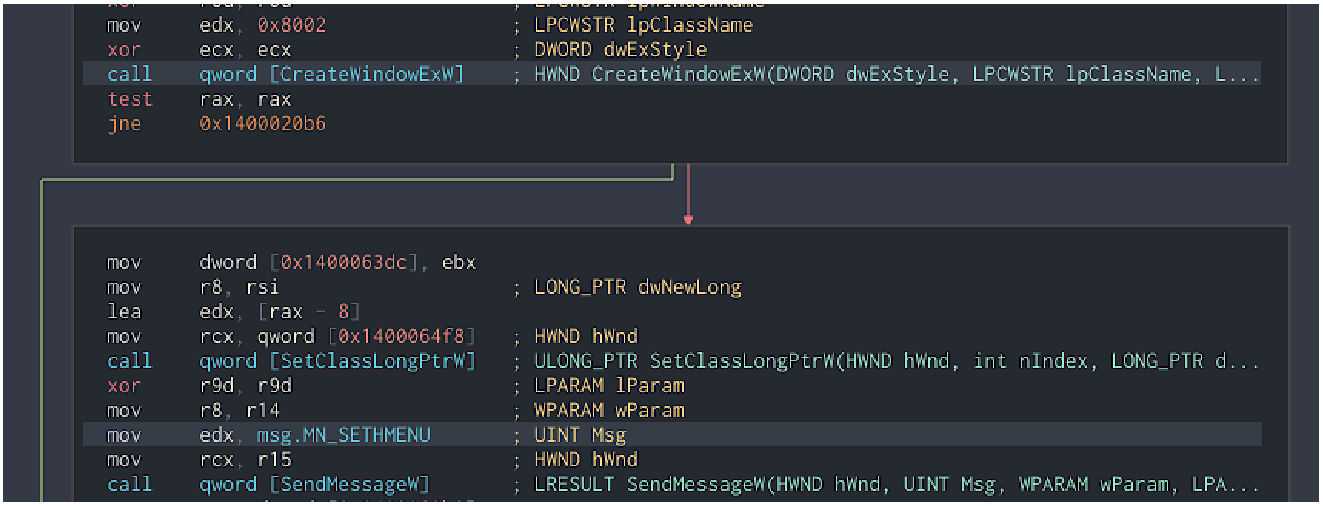
Research by: Itay Cohen, Eyal Itkin

In the past months, our Vulnerability and Malware Research teams joined efforts to focus on the exploits inside the malware and specifically, on the exploit writers themselves. Starting from a single Incident Response case, we built a profile of one of the most active exploit developers for Windows, known as "*Volodya*" or "*BuggiCorp*". Up until now, we managed to track down more than 10 (!) of their Windows Kernel Local Privilege Escalation (LPE) exploits, many of which were zero-days at the time of development.

Background

Our story begins, as all good stories do, with an incident response case. When analyzing a complicated attack against one of our customers, we noticed a very small 64-bit executable that was executed by the malware. The sample contained unusual debug strings that pointed at an attempt to exploit a vulnerability on the victim machine. Even more importantly, the sample had a leftover PDB path which proclaimed loud and clear the goal of this binary: `...\cve-2019-0859\x64\Release\CmdTest.pdb`. With the absence of any online resource with this implementation of CVE-2019-0859, we realized that we are not looking at a publicly available PoC, but rather a real-world exploitation tool. This intrigued us to dig deeper.

Reverse-engineering the exploit was pretty straight forward. The binary was small, and the debug messages were there to guide us. It exploited a use-after-free (UAF) vulnerability in `CreateWindowEx` to gain elevated privileges to the parent process. We quickly made an interesting observation: it seemed like the exploit and the malware itself weren't written by the same people. The code quality, lack of obfuscation, PDBs, and timestamps, all pointed to this conclusion.



```
mov     edx, 0x8002                ; LPCWSTR lpClassName
xor     ecx, ecx                   ; DWORD dwExStyle
call    qword [CreateWindowExW]    ; HWND CreateWindowExW(DWORD dwExStyle, LPCWSTR lpClassName, L...
test    rax, rax
jne     0x1400020b6

mov     dword [0x1400063dc], ebx
mov     r8, rsi                    ; LONG_PTR dwNewLong
lea     edx, [rax - 8]
mov     rcx, qword [0x1400064f8]   ; HWND hWnd
call    qword [SetClassLongPtrW]   ; ULONG_PTR SetClassLongPtrW(HWND hWnd, int nIndex, LONG_PTR d...
xor     r9d, r9d                   ; LPARAM lParam
mov     r8, r14                    ; WPARAM wParam
mov     edx, msg.MN_SETHMENU       ; UINT Msg
mov     rcx, r15                   ; HWND hWnd
call    qword [SendMessageW]       ; LRESULT SendMessageW(HWND hWnd, UINT Msg, WPARAM wParam, LPA...
```

Figure 1: A call to `CreateWindowEx`, as can be seen in Cutter.

Exploits Distribution 101

We tend to look at the people behind a specific malware family as one unbroken unit. It's easier to envision that each and every component was written by a single person, team, or group. Truth is, writing an advanced malware, by nation-states or criminals, involves different groups of people with various skills. A cyber-espionage organization of a nation-state, is likely to have hundreds or even thousands of employees in different groups and branches. Each worker in the organization has a specific role, fine-tuned by special technological training and years of expertise. In such an organization, the workload of writing the common components is broken down among specialized teams, with different ones responsible for the initial access, collecting sensitive data, lateral movement, and more.

An operational entity whose goal is to embed an exploit module in its malware can't rely on malware developers alone. Finding a vulnerability, and reliably exploiting it, will most probably be done by specific teams or individuals who specialize in a particular role. The malware developers for their part don't really care how it works behind the scenes, they just want to integrate this module and be done with it.

For this division of labor to work, both teams need to agree on some API that will be the bridge between the different components. This integration API isn’t unique to state actors, but is a common feature in the “free market” of exploits. Whether it involves underground forums, exploit brokers, or offensive cyber companies, they all provide their customers with instructions on how to integrate the exploit in their malware.

In essence, this integration point is the key aspect that we would like to focus on in our research. Assuming that exploit authors work independently, and only distribute their code/binary module to the malware authors, we decided to focus on them for a change. By analyzing the exploits embedded in malware samples, we can learn more about the exploit authors, hopefully distinguishing between them by studying their coding habits and other fingerprints left as clues on their identity, when distributing their products to their malware writing counterparts.

Fingerprinting Exploit Developers

Instead of focusing on an entire malware and hunting for new samples of the malware family or actor, we wanted to offer another perspective and decided to concentrate on these few functions that were written by an exploit developer. Having this small 64-bit binary from the incident response case looked like a promising start.

The binary did nothing other than exploiting CVE-2019-0859 and wasn’t based on a source-code or a POC that was shared publicly. It made a great candidate for us to fingerprint, as the executable was refined from code written by someone other than the exploit author. Moreover, the executable was separated from the main binary of the malware, an infamous crimeware, which made us believe that this exploit wasn’t developed in-house by the malware developers. With this hope, we set out to find more exploits written by the same author.

We started by gathering simple artifacts from the binary we already had: strings, internal file name, timestamps, and the PDB path. The first result came immediately — a 32-bit executable that was an exact match to the 64-bit sample. Specifically, as their timestamps and embedded PDB path showed, they were compiled together, at the same time and from the same source code. Now that we had these two samples, we were able to formulate what we should look for.

To fingerprint the author of this exploit, we set our sights on the following:

- Unique artifacts in the binaries
    - Hard-coded values (Crypto constants, “garbage” values such as 0x11223344)
    - Data tables (Usually version-specific configurations)
    - Strings (GDI object names: “MyWindow”, “MyClass_56”, “findme1”, …)
    - PDB path

- Code snippets
    - Unique implementation of functions
        - Syscall wrappers
        - Inline assembly
        - Proprietary crypto functions / implementations
    - Techniques and habits
        - Preferred leaking technique (`HMValidateHandle` , `gSharedInfo` , etc.)
        - Preferred elevation technique (How is the token replacement performed?)
        - Heap spraying technique (Using AcceleratorTables? Windows? Bitmaps?)
    - Framework
        - The flow of the exploits
            - **Option #1:** Main exploit flow with almost no side-branches
            - **Option #2:** Multiple twists and knobs for different versions of the OS
        - The structure of the code and functions in it
            - **Modularity:** Separation to functions
            - **Structure:** Separation to clear phases (Init, config, spray, token swap, …)
            - **Global Variables:** What information is stored in global variables? (OS version? OS version enum? Just a specific field offset?)
        - Version-specific configurations:
            - **Field offsets:** What fields are of special interest?
            - **Preferred system calls:** Preferred set of syscalls
        - API provided to the customer

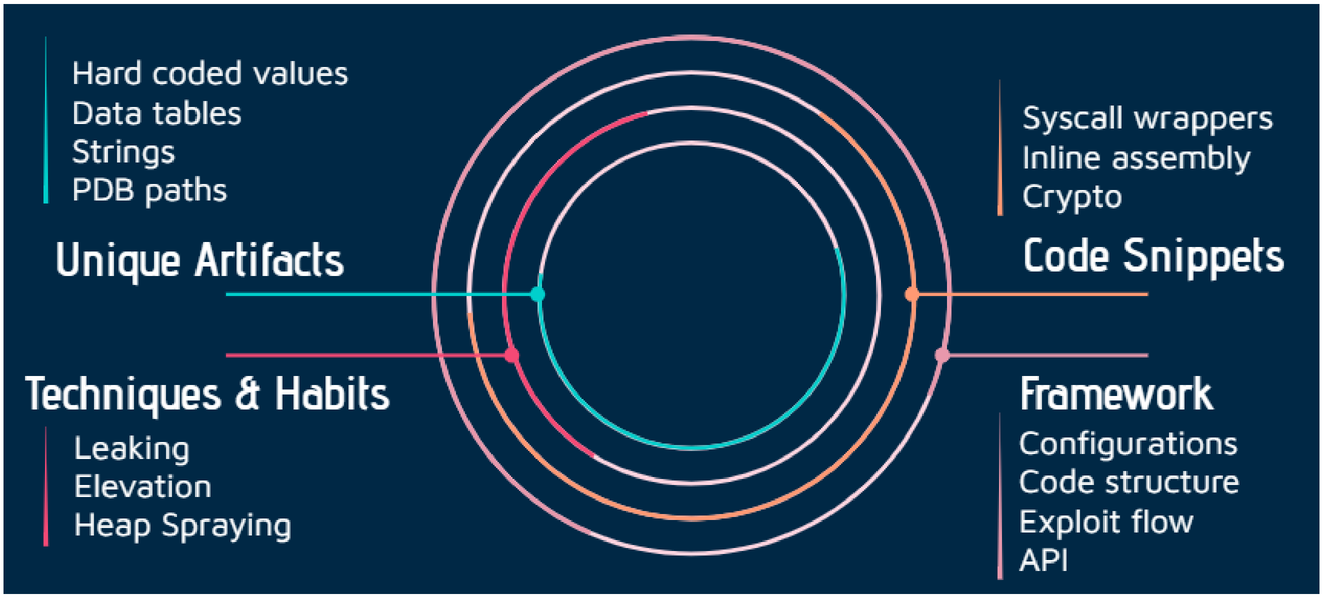


Figure 2: The set of exploit-related artifacts that we will be looking for.

With these properties in mind, we looked back at the two samples we had and marked some artifacts we thought were unique. Even though we had only two small binaries (which were essentially the same) we were able to create hunting rules to find more samples written by this developer. To our surprise, we were able to find more of them than we could have imagined.

One after the other, dozens of samples started to appear, and with each one, we improved our hunting rules and methodologies. With a careful analysis of the samples, we were able to understand which samples exploited which CVE, and based on that created a timeline to understand whether the exploit was written as a 0-day before it was exposed, or was it a 1-day that was implemented based on patch-diffing and similar techniques.

At this point, we had more than ***10 CVEs*** that we were able to attribute to the same exploit developer, based on our fingerprinting technique alone and without further intelligence. Later on, public reports revealed the name of our target exploit seller: *Volodya* (a.k.a Volodimir), previously known as *BuggiCorp*. It seemed we were not the only ones to track this exploit seller, as Kaspersky reported some relevant information about them on several occasions. In addition, ESET also mentioned some of Volodya's incriminating trails in their VB2019 talk about Buhtrap.

According to Kaspersky, Volodya first made headlines under their "BuggiCorp" nickname, when they advertised a Windows 0-day for sale on the infamous Exploit[.]in cyber-crime forum with a starting price of $95,000. Across the years, the price went up and some of their Windows LPE 0-day exploits were sold at a price as high as $200,000. As published in Kaspersky's report, and later confirmed by us, Volodya sold exploits to both crimeware and APT groups. We discuss the actor's clients in more detail under the chapter "The Customers".

Our actor's exploits

Although a few of our initial hunting rules needed some fine-tuning, even the immediate results we received were quite surprising. After further calibration, we managed to find numerous samples, all of which were Local Privilege Escalation (LPE) exploits in Windows. Out of these samples, we were able to identify the following list of CVEs that were exploited by our actor.

Side note:

*During the classification of the exploits, we chose to take a conservative approach when deciding if a given vulnerability was exploited as a 0-Day or 1-Day. If other security vendors attributed the in-the-wild exploit to our actor, then it was a 0-Day. If we found sufficient evidence that one of our samples is indeed the exploit circulating in the wild, **exactly** as was described by a vendor in their report, then we also flagged it as such.*

In all other cases, we marked the vulnerability as an exploited 1-Day, preferring to have a lower bound of the 0-Day count instead of mistakenly overshooting the correct number.

CVE-2015-2546

Classification: 1-Day
Basic Description: Use-After-Free in `xxxSendMessage` (`tagPOPUPMENU`)
0-Day vendor report: FireEye
Found in the following Malware samples: Ursnif, Buhtrap

Our exploit samples use a different memory shaping technique than the one described in the initial report: spraying Windows instead of Accelerator Tables. In addition, our earliest and most basic exploit sample contains the following PDB path, suggesting the author already knew the CVE-ID for this vulnerability: “C:\…**volodimir_8**\c2**CVE-2015-2546**_VS2012\x64\Release**CmdTest**.pdb”

CVE-2016-0040

Classification: 1-Day
Basic Description: Uninitialized kernel pointer in `WMIDataDevice IOControl`
0-Day vendor report: N/A. Was never exploited as a 0-Day in the wild
Found in the following Malware samples: Ursnif

This exploit was used in a single sample that also contained the previously described exploit for CVE-2015-2546. This exploit is selected if the target is a Windows version earlier than Windows 8. Otherwise, CVE-2015-2546 is used.

CVE-2016-0167

Classification: 0-Day
Basic Description: Use-After-Free in `Win32k!xxxMNDestroyHandler`
0-Day vendor report: FireEye.
Found in the following Malware samples: *PUNCHBUGGY*

Our exploit samples align perfectly with the technical report about the in-the-wild exploit.

CVE-2016-0165*

Classification: 1-Day
Basic Description: Use-After-Free in `Win32k!xxxMNDestroyHandler`
0-Day vendor report: Found by **Kaspersky**, but no report was published publicly
Found in the following Malware samples: Ursnif

This is an interesting case. Our actor’s 0-Day (CVE-2016-0167) was patched by Microsoft in April 2016. The same patch also fixed CVE-2016-0165 which was also used in the wild. Searching for a new vulnerability to exploit, our actor probably patch-diffed Microsoft’s fixes and found a vulnerability that they thought was the patched 0-Day. This vulnerability originates in the patched function used in their previous vulnerability: `Win32k!xxxMNDestroyHandler`.

*We have multiple indications from their exploit samples for this vulnerability that either the exploit author or at least their customers were certain that they were sold an exploit for CVE-2016-0165. The sad truth is, after analyzing the exploit, we can say that the exploited vulnerability is a different one.

```
push    str.LdrCveElevate           ; "LdrCveElevate"
mov     ecx,    edi
sub     ecx,    eax
push    str.Trying_CVE_2016_0165_exploit ; "[%s:%u] Trying CVE_2016_0165 exploit...\n"
mov     dword [ebp - 4], eax
push    ecx
lea     eax,    [ebp + eax - 0x218]
push    eax
call    esi                         ; snprintf
add     dword [ebp - 4], eax
add     esp,    0x2c
push    dword [ebp - 4]
lea     eax,    [ebp - 0x218]
push    eax
call    fcn.10007f12
push    dword [0x1000e674]
call    exploit
mov     dword [ebp - 4], eax
test    eax,    eax
lea     eax,    [ebp - 0x14]
push    eax
je      0x100013e5
```

Figure 3: Debug string indicating the confusion around CVE-2016-0165, as can be seen in Cutter.

This confusion is probably due to the fact that Microsoft releases a single fix that addresses multiple vulnerabilities, and they are the only ones with the full mapping between each code fix and the CVE that was issued for it.

CVE-2016-7255

Classification: 0-Day
Basic Description: Memory corruption in `NtUserSetWindowLongPtr`
0-Day vendor report: Reported by Google, a technical report by TrendMicro
Found in the following Malware samples: Attributed to *APT28* (aka Fancy Bear, Sednit). Used later by Ursnif, Dreambot, GandCrab, Cerber, Maze

Our exploit samples align perfectly with the technical report about the in-the-wild exploit. This specific exploit was later widely used by different ransomware actors. In addition, we've seen other exploits for this specific vulnerability that were sold as 1-Days to other ransomware actors as well.

Note: We have multiple circumstantial evidence to believe that this 0-Day was the one that was mentioned by BuggiCorp in the famous ad posted to the exploit[.]in forum in May 2016.

CVE-2017-0001

Classification: 1-Day
Basic Description: Use-After-Free in `RemoveFontResourceExW`
0-Day vendor report: N/A. Was never exploited as a 0-Day in the wild
Found in the following Malware samples: Attributed to *Turla*. Later used by Ursnif

Used as a 1-Day in operations attributed to Turla (FireEye).

CVE-2017-0263

Classification: 0-Day
Basic Description: Use-After-Free in `win32k!xxxDestroyWindow`
0-Day vendor report: ESET
Found in the following Malware samples: Attributed to *APT28* (aka Fancy Bear, Sednit)

Our exploit samples align perfectly with the technical report about the in-the-wild exploit.

CVE-2018-8641*

Classification: 1-Day
Basic Description: Double Free in `win32k!xxxTrackPopupMenuEx`
0-Day vendor report: N/A. Was never exploited as a 0-Day in the wild
Found in the following Malware samples: Magniber

Once again, identifying the used 1-Days is usually harder than identifying 0-Days. This time, we couldn't find any sample that might hint as to what was the vulnerability the actor thought they were exploiting.

*We identified that this specific vulnerability was patched by Microsoft in December 2018. After scanning the list of vulnerabilities that were addressed in this patch, we are pretty certain that Microsoft labeled this vulnerability as CVE-2018-8641, but we can't know for sure.

Update: On June 24, 2020 Kaspersky published in their blog an analysis of exploits distributed through the Magnitude exploit kit. In their blog post, Kaspersky analyzed the LPE exploit used by Magniber, attributed it to Volodya and estimated it is probably CVE-2018-

8641. This independent conclusion on behalf of Kaspersky strengthens our initial estimate.

CVE-2019-0859

Classification: 0-Day
Basic Description: Use-After-Free in `CreateWindowEx`
0-Day vendor report: Kaspersky
Found in the following Malware samples: Used as a standalone component to be injected or loaded. We couldn't attribute it to any specific APT/malware.

Our exploit samples align perfectly with the technical report about the in-the-wild exploit. Our research started with a single sample of this exploit that was found in a customer's network. In one of the samples we found later on, we could see this clear PDB string: "X:\tools**0day****09-08-2018**\x64\Release\RunPS.pdb", as opposing to the PDB string in our initial sample: "S:\Work\Inject**cve-2019-0859**\Release\CmdTest.pdb".

CVE-2019-1132*

Classification: 0-Day
Basic Description: NULL pointer dereference at `win32k!xxxMNOpenHierarchy` (`tagPOPUPMENU`)
0-Day vendor report: ESET
Found in the following Malware samples: Attributed to *Buhtrap*

*We have multiple reasons to believe that this was another 0-Day exploit from Volodya, as multiple technical details in the report match their typical ways of exploitation. In addition, the exploit reported having the following PDB path embedded in it: "C:\work**volodimir_65**\…pdb". However, this is the only exploit in our list that we have not yet found a sample of, and so we can't be sure in our attribution for this exploit.

CVE-2019-1458

Classification: 1-Day
Basic Description: Memory corruption in window switching
0-Day vendor report: Kaspersky (Initial Report, Detailed Report)
Found in the following Malware samples: Attributed to operation *WizardOpium*

Our exploit doesn't align with the technical report about the in-the-wild exploit. In addition, in their detailed report, Kaspersky noted that "it was also interesting that we found another 1-day exploit for this vulnerability just one week after the patch, indicating how simple it is to exploit this vulnerability." And indeed, our sample is dated to 6 days after Kaspersky's initial report.

Vulnerabilities Summary

Here is a table summarizing the vulnerabilities we've listed:

CVE	**Is 0-Day?**
CVE-2015-2546	No
CVE-2016-0040	No
CVE-2016-0165*	No
CVE-2016-0167	**Yes**
CVE-2016-7255	**Yes**
CVE-2017-0001	No
CVE-2017-0263	**Yes**
CVE-2018-8641*	No
CVE-2019-0859	**Yes**
CVE-2019-1132*	**Yes**
CVE-2019-1458	No
Total Count	**5** out of **11**

The author's fingerprints

Now that we found more than 10 different exploits from Volodya, we can review them in greater detail and familiarize ourselves with the actor's work habits. It was clear to us from the beginning that our actor probably has a simple template they deploy for the different exploits, as the function flow of each exploit, and even the order of the different functions, were shared between most of the exploits.

Throughout this section, we describe a collection of key characteristics, that reflect the different implementation choices made by Volodya when creating the exploit template. We compare their implementation to that of another exploit writer, known by the nickname *PlayBit*. By this comparison we aim to outline the wide variety of implementation options that are present in each part of the exploit, making each author's set of implementation choices a unique "signature" of their way of thinking and working.

PlayBit (a.k.a luxor2008)

Using the same technique we used to hunt Volodya's exploits, we managed to track down 5 Windows LPE 1-Day exploits that were written by PlayBit, in addition to other tools that the author sold throughout the years. We started from a single sample, CVE-2018-8453 which is used by REvil ransomware, and used PlayBits' unique fingerprints to seek out more exploits.

We found the following Windows LPE exploits implemented as 1-days by this author:

- CVE-2013-3660
- CVE-2015-0057
- CVE-2015-1701
- **CVE-2016-7255 –** This is a 0-Day of Volodya
- CVE-2018-8453

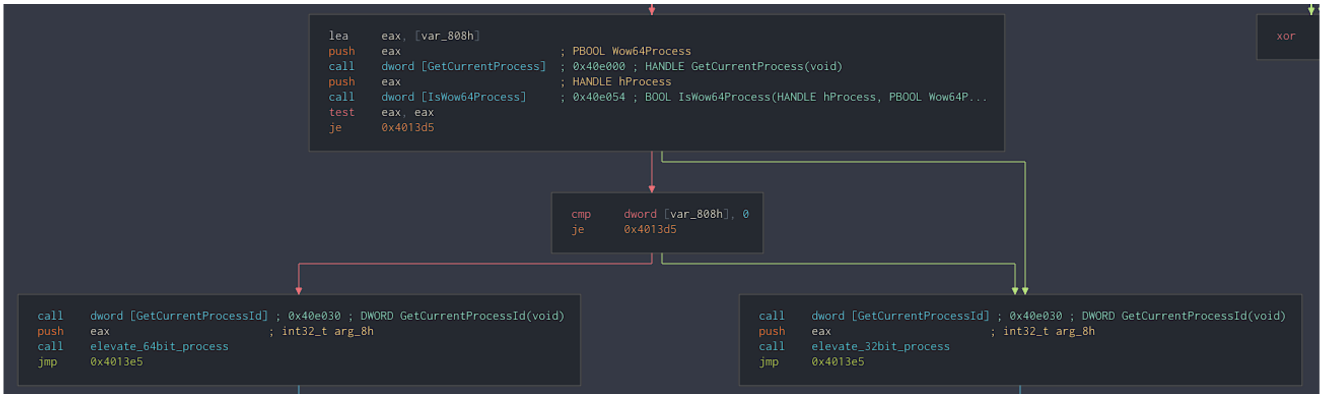
Technically, PlayBit also sold two exploits for CVE-2019-1069 (a SandboxEscaper vulnerability) and CVE-2020-0787. However, we ignore these exploits as they aren't memory corruption vulnerabilities, but rather a vulnerability in different services, and as such have a different structure.

Note: A deeper analysis of PlayBit, and the different exploits they developed and sold, will be released in an upcoming blog post.

bool elevate(int target_pid)

The API in all of Volodya's exploit samples is always the same. Regardless of whether it was embedded inside a malware sample, or was a standalone POC, the exploit had a single API function of the following signature:

`bool elevate(int target_pid)`



```
lea     eax, [var_808h]
push    eax                          ; PBOOL Wow64Process
call    dword [GetCurrentProcess]    ; 0x40e000 ; HANDLE GetCurrentProcess(void)
push    eax                          ; HANDLE hProcess
call    dword [IsWow64Process]       ; 0x40e054 ; BOOL IsWow64Process(HANDLE hProcess, PBOOL Wow64P...
test    eax, eax
je      0x4013d5
```

```
cmp     dword [var_808h], 0
je      0x4013d5
```

```
call    dword [GetCurrentProcessId] ; 0x40e030 ; DWORD GetCurrentProcessId(void)
push    eax                         ; int32_t arg_8h
call    elevate_64bit_process
jmp     0x4013e5
```

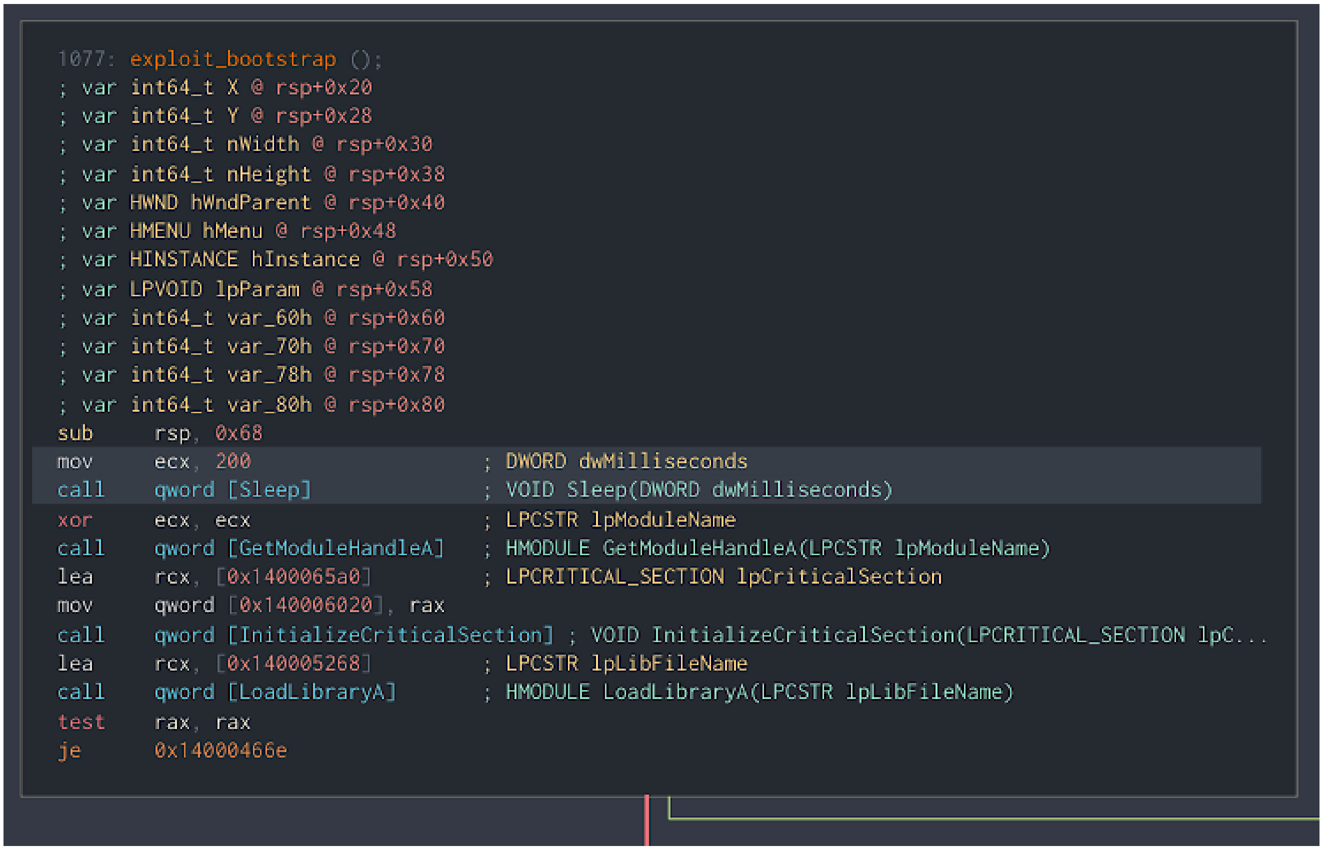
```
call    dword [GetCurrentProcessId] ; 0x40e030 ; DWORD GetCurrentProcessId(void)
push    eax                         ; int32_t arg_8h
call    elevate_32bit_process
jmp     0x4013e5
```

Figure 4: Invoking the elevate(target_pid) function, as can be seen in Cutter.

The exploit itself doesn’t include any feature for injecting shellcode into another process or anything fancy of this sort. It grants SYSTEM privileges to the desired process, taking nothing other than its PID as an argument.

Sleep(200)

The very first thing that the `elevate()` function does, right after it’s invoked by the malware, is to `Sleep()` for a constant time period of 200 milliseconds.



```
1077: exploit_bootstrap ();
; var int64_t X @ rsp+0x20
; var int64_t Y @ rsp+0x28
; var int64_t nWidth @ rsp+0x30
; var int64_t nHeight @ rsp+0x38
; var HWND hWndParent @ rsp+0x40
; var HMENU hMenu @ rsp+0x48
; var HINSTANCE hInstance @ rsp+0x50
; var LPVOID lpParam @ rsp+0x58
; var int64_t var_60h @ rsp+0x60
; var int64_t var_70h @ rsp+0x70
; var int64_t var_78h @ rsp+0x78
; var int64_t var_80h @ rsp+0x80
sub     rsp, 0x68
mov     ecx, 200                   ; DWORD dwMilliseconds
call    qword [Sleep]              ; VOID Sleep(DWORD dwMilliseconds)
xor     ecx, ecx                   ; LPCSTR lpModuleName
call    qword [GetModuleHandleA]   ; HMODULE GetModuleHandleA(LPCSTR lpModuleName)
lea     rcx, [0x1400065a0]         ; LPCRITICAL_SECTION lpCriticalSection
mov     qword [0x140006020], rax
call    qword [InitializeCriticalSection] ; VOID InitializeCriticalSection(LPCRITICAL_SECTION lpC...
lea     rcx, [0x140005268]         ; LPCSTR lpLibFileName
call    qword [LoadLibraryA]       ; HMODULE LoadLibraryA(LPCSTR lpLibFileName)
test    rax, rax
je      0x14000466e
```

Figure 5: Starting the exploit with a call to `Sleep(200)`, as can be seen in Cutter.

It is not absolutely clear why the `Sleep(200)` is there in the template of the exploits. We suspect it is to avoid unnecessary instability, especially because most of these exploits are based on timing (UAF, races). Therefore, waiting a short while for the I/O and memory access related activities to end could improve stability. As the exploits are part of malware, all this malware-related code before the exploit's execution will cause a short spike in CPU/disk/RAM, and it might make sense to let things calm down a bit before moving on to the actual exploit. For short-term spike load (that naturally occurs when starting new processes, reading/writing files from disk, etc.), it should be enough to wait 200ms.

Although we've noticed a change in this pattern in the most recent samples, this feature can still be found in 9 of the exploits we've seen.

Comparison to PlayBit: PlayBit doesn't have any such feature in their exploits.

OS Fingerprinting

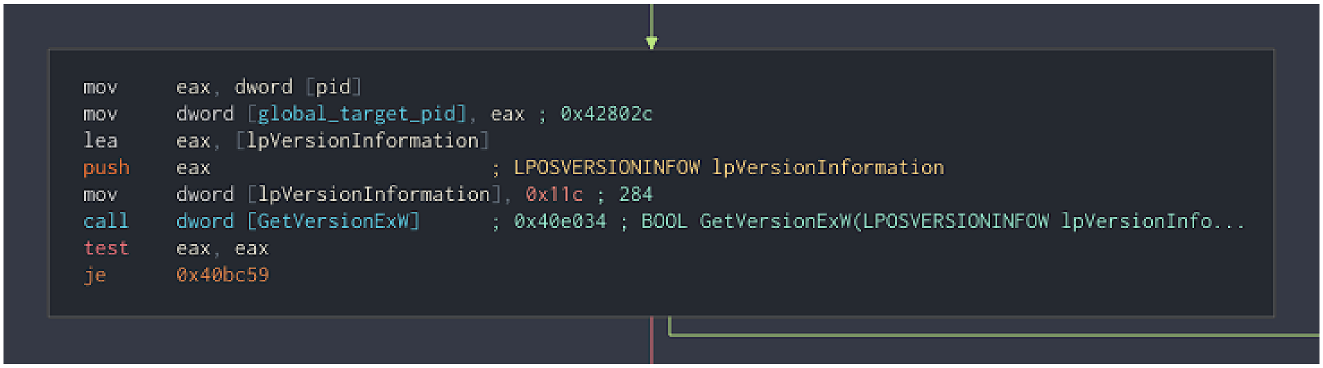
Right after waking from its beauty sleep, the exploit identifies and calibrates itself to the target's Windows version, so as to facilitate the support for as many OS versions as possible. From our samples, it seems that the author queries the OS using two techniques:

Parsing ntdll.dll's header

This is the most commonly used technique. A handle into `ntdll.dll` is used to find the offset into the `IMAGE_NT_HEADERS`, from which the `MajorOperatingSystemVersion` and the `MinorOperatingSystemVersion` fields are parsed.

GetVersionEx()

This technique is usually used together with the previous one and was only used in samples from 2016 to the beginning of 2017. This is probably due to the fact that this API is now deprecated.



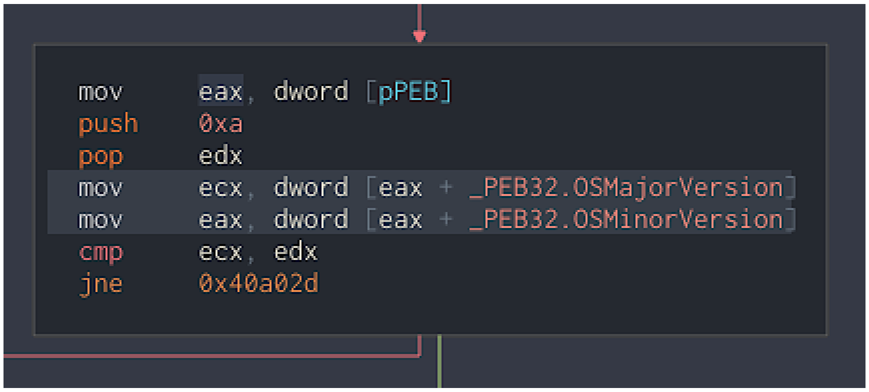
```
mov     eax, dword [pid]
mov     dword [global_target_pid], eax ; 0x42802c
lea     eax, [lpVersionInformation]
push    eax                          ; LPOSVERSIONINFOW lpVersionInformation
mov     dword [lpVersionInformation], 0x11c ; 284
call    dword [GetVersionExW]        ; 0x40e034 ; BOOL GetVersionExW(LPOSVERSIONINFOW lpVersionInfo...
test    eax, eax
je      0x40bc59
```

Figure 6: Calling `GetVersionExW()` to get Windows's version, as can be seen in Cutter.

In both of these techniques, the goal is to query both the major and minor version of the OS, and configure the exploit's global variables accordingly.

While most exploits support a wide range of Windows versions, Volodya never seems to care about the specific service pack of the target, nor about whether it is a Windows server or not. Aside from the interest in specific Windows 10 build versions, used only in the exploit for CVE-2019-1458, our actor only uses the major and minor versions, and that’s it.

Comparison to PlayBit: Once again, `GetVersionEx()` is used, usually with a later additional parsing of the major and minor numbers from the Process Environment Block (PEB) itself, as can be seen in Figure 7. Not only is PEB used instead of `ntdll.dll`, PlayBit also extracts additional information from the `GetVersionEx()` output such as the computer’s Service Pack, and even checks if the target computer uses a server operating system.



```
mov     eax, dword [pPEB]
push    0xa
pop     edx
mov     ecx, dword [eax + _PEB32.OSMajorVersion]
mov     eax, dword [eax + _PEB32.OSMinorVersion]
cmp     ecx, edx
jne     0x40a02d
```

Figure 7: Extracting the major and minor versions from the PEB, as can be seen in Cutter.

This is a clear difference in the modus operandi of both actors. Not only do they extract the same information in different ways, Volodya is interested in far less information than PlayBit, even when they both exploit the same vulnerability (CVE-2016-7255).

In general, both actors hold detailed version-specific configurations from which they load the relevant information once the OS version is determined. The main difference between the two is that the code flow in Volodya’s exploits rarely depends on the OS version, while PlayBit incorporates multiple twists and knobs using various if-checks that depend on the OS version. This in turn affects their different interest in the exact version details.

Leaking Kernel Addresses

In the vast majority of exploits, the actor tunes the exploit using a kernel-pointer-leak primitive. In all exploits except CVE-2019-1458, this leak primitive is the well-known HMValidateHandle technique.

`HMValidateHandle()` is an internal unexported function from `user32.dll`, that is leveraged by various functions such as `isMenu()`, and can be used to get the kernel address of different Window objects in all Windows versions up to Windows 10 RS4. This technique was well known and used even back in 2011, whereby most exploitation tutorials chose to specifically parse `isMenu()` to find the address of `HMValidateHandle()`.

It is surprising to see that out of dozens of different functions that could be used for finding `HMValidateHandle()`, the actor simply followed the well-known tutorials and chose to use `isMenu()` as well. It is even more surprising to see that this common exploitation technique still worked quite well throughout the years, giving the actor no incentive to try to "hide" by picking a less known function such as `CheckMenuRadioItem()`.

The leak gives us the following:

- Kernel address of our window.
- Kernel address of our `THREAD_INFO` (the `pti` field).

This information is used in several steps during the exploit:

- Addresses are used when pointing to / creating fake kernel structs.
- Making sure our kernel address is a valid Unicode string (doesn't contain two consecutive '\x00' bytes).
- The `pti` is used to locate a valid EPROCESS, which is then used during the Token Swap phase.

Comparison to PlayBit: PlayBit chose to implement this feature via direct access to the user-mode Desktop Heap. More on this subject could be found in the future blog post focusing on this actor.

Token Swap

The ultimate goal of the exploit is to grant SYSTEM privileges to the desired process, according to the given PID argument. Traditionally, the way to achieve this is by replacing the process's token in the EPROCESS/KPROCESS structure with the token of the SYSTEM process.

Here are some common techniques for doing exactly that. You'd be surprised to see how many different options there are for implementing this feature.

Using Ps* symbols

The Windows kernel contains the following functions and global variables for process-related functionality:

- `PsLookupProcessByProcessId` – Retrieves a pointer to the process's EPROCESS.
- `PsInitialSystemProcess` – Global variable holding a pointer to the SYSTEM's EPROCESS.
- `PsReferencePrimaryToken` – Returns a pointer to the primary token of the process.

By executing these functions in kernel-mode, a given shellcode can easily locate SYSTEM's token, but it still doesn't solve the issue of how to assign it in the required EPROCESS.

For this purpose there are 2 common solutions:

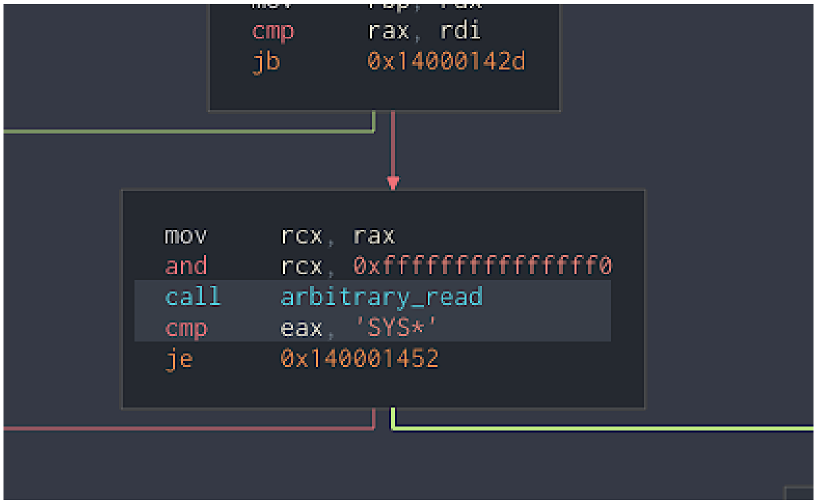
- Directly access the correct offset inside the EPROCESS using a version-specific offset.
- Scan the EPROCESS in search of our own pointer (known by the previous call to `PsReferencePrimaryToken`) and replace the entry once a match is found.

This technique requires executing code in kernel-mode, and so will be blocked by the SMEP protection, unless an additional SMEP bypass is deployed.

Scanning the PsList

The common alternative for locating the EPROCESS of both the target and SYSTEM processes, is to scan the doubly-linked process list, referred to as PsList. The steps involved in this technique are:

1. Locate an initial EPROCESS (using the leaked pti field).
2. Scan the PsList in search of an EPROCESS with the target PID.
3. Scan the PsList in search of the EPROCESS of SYSTEM by looking for a PID of 4, or a name of `SYS*` .
4. Extract the token and place it in the matching offset in the target process.
5. Cautiously update the reference count of SYSTEM's token.



```
cmp     rax, rdi
jb      0x14000142d
```

```
mov     rcx, rax
and     rcx, 0xfffffffffffffff0
call    arbitrary_read
cmp     eax, 'SYS*'
je      0x140001452
```

Figure 8: Volodya exploit using an Arbitrary-Read primitive in search for `SYS*` , as can be seen in Cutter.

This technique requires the offset to both the primary token **and** the `LIST_ENTRY` for the PsList, pretty much mandating that they are both stored as part of a version-specific configuration.

The major advantage of this technique is that while it can still be executed as a simple shellcode in kernel-mode (as done in the exploit of CVE-2017-0263), it can also be implemented completely in user-mode. To do so, you need two exploit primitives, one for an Arbitrary-Read (from kernel-space) and the other for an Arbitrary-Write (into kernel-space). Running in user-mode solves the issues we detailed before in regards to SMEP, rendering this protection useless against such exploit primitives.

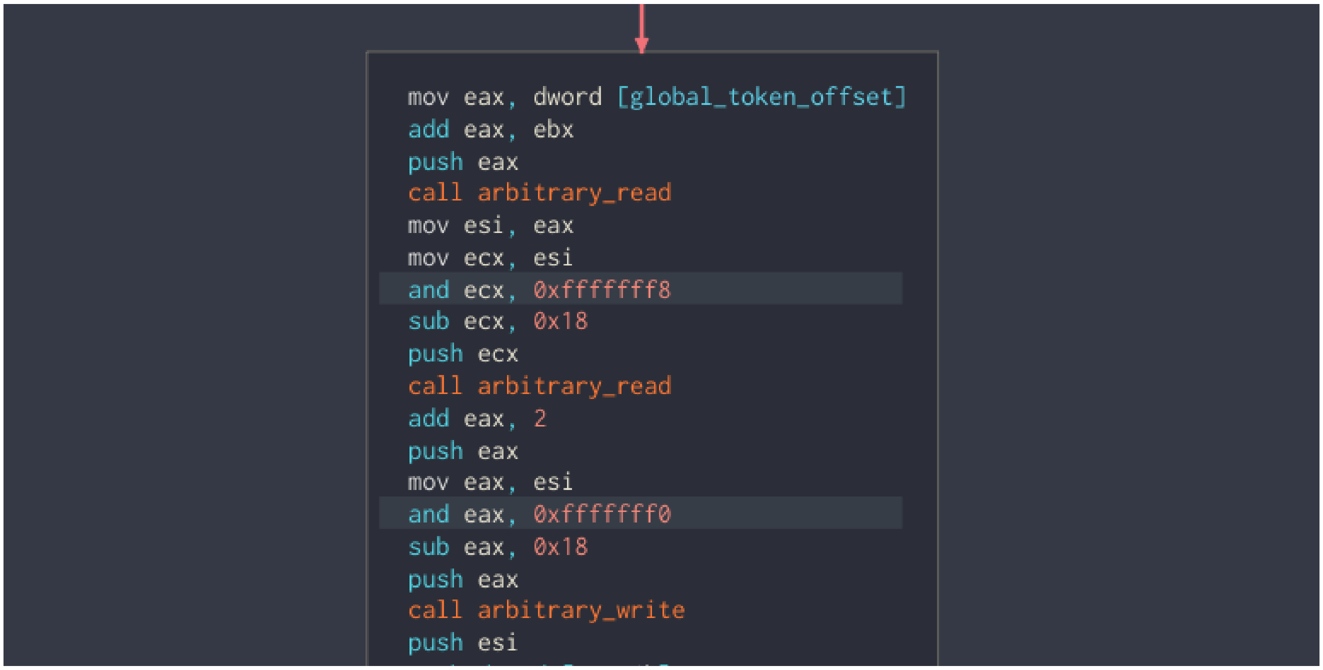
As the token is a reference-counted object, it is important to properly register the reference that was just added so as to avoid a Blue-Screen-Of-Death (BSOD) when the elevated process terminates. In fact, there are two different reference counts:

- The token is an `EX_FAST_REF` object – the lower pointer bits are used as a ref-count.
- An `OBJECT_HEADER` is stored before the token, holding yet another ref-count.

As out actor chose to update the latter ref-count field, the following steps will be needed:

1. Mask out the ref-count bits from the token's pointer – should be aligned to 8 bytes on 32-bit processes, and 16 bytes on 64-bit processes.
2. Subtract the constant needed in order to point at `OBJECT_HEADER`'s ref-count field.
3. Read the value (using an Arbitrary-Read exploit primitive).
4. Increment it accordingly.
5. Write back the updated value.

However, as can be seen in Figure 9, we found the following bug in all of the 32-bit exploits that contained this feature:



```
mov eax, dword [global_token_offset]
add eax, ebx
push eax
call arbitrary_read
mov esi, eax
mov ecx, esi
and ecx, 0xfffffff8
sub ecx, 0x18
push ecx
call arbitrary_read
add eax, 2
push eax
mov eax, esi
and eax, 0xfffffff0
sub eax, 0x18
push eax
call arbitrary_write
push esi
```

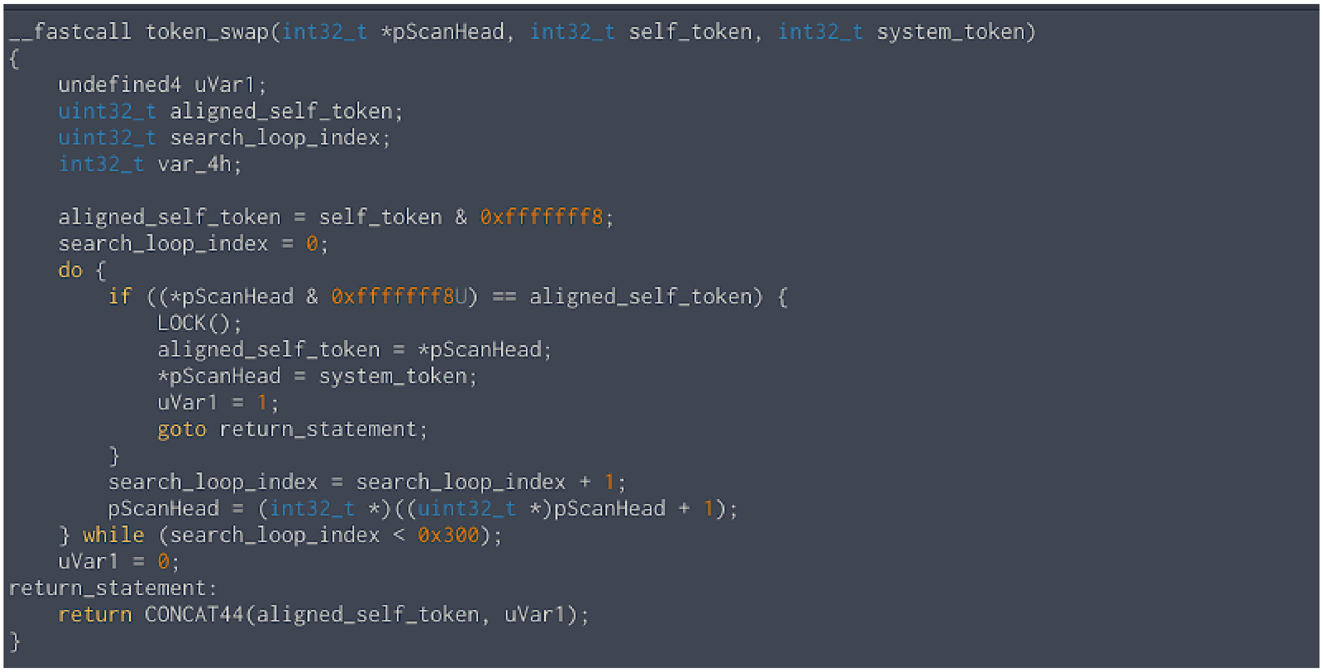
Figure 9: An implementation bug in the reference-count update used in 32-bit exploits.

The alignment mask when reading the reference-count value is an alignment to 8 bytes, while a **different** mask is used when writing back the updated value. If the token will be stored in a memory address that is aligned to 8 bytes and is not aligned to 16 bytes, the write operation will update the wrong field.

While CVE-2016-0040 and CVE-2016-0167 use the Ps* technique, scanning the PsList is by-far our actor's favorite way of performing a token swap, used in 8 of their exploits. In 7 of these, they used Arbitrary-Read and Arbitrary-Write from user-mode.

Comparison to PlayBit: In all of their samples, we've always seen PlayBit use the Ps* functions for a token swap. This decision forced the actor to implement a few SMEP bypasses they integrated into their later exploits for CVE-2016-7255 and CVE-2018-8453. This design choice explains why the actor doesn't bother implementing a proper Arbitrary-Read primitive as part of the exploit. Instead of using a version-specific configuration for the offset of the token in the EPROCESS, PlayBit always scans the EPROCESS to search for it, usually using 0x300 or 0x600 as the upper limit for the search.

It is worth noting that the memory corruption technique that is used by PlayBit in the different exploits was also used by Duqu 2.0 and was analyzed in Microsoft's previous VB talk from 2015. Through this memory corruption, they can trigger a few memory read/writes from/to Kernel memory that will help during the exploit.



```
__fastcall token_swap(int32_t *pScanHead, int32_t self_token, int32_t system_token)
{
    undefined4 uVar1;
    uint32_t aligned_self_token;
    uint32_t search_loop_index;
    int32_t var_4h;

    aligned_self_token = self_token & 0xfffffff8;
    search_loop_index = 0;
    do {
        if ((*pScanHead & 0xfffffff8U) == aligned_self_token) {
            LOCK();
            aligned_self_token = *pScanHead;
            *pScanHead = system_token;
            uVar1 = 1;
            goto return_statement;
        }
        search_loop_index = search_loop_index + 1;
        pScanHead = (int32_t *)((uint32_t *)pScanHead + 1);
    } while (search_loop_index < 0x300);
    uVar1 = 0;
return_statement:
    return CONCAT44(aligned_self_token, uVar1);
}
```

Figure 10: PlayBit exploit scanning the EPROCESS in search for the token, as can be seen in Cutter.

Wrapping it up

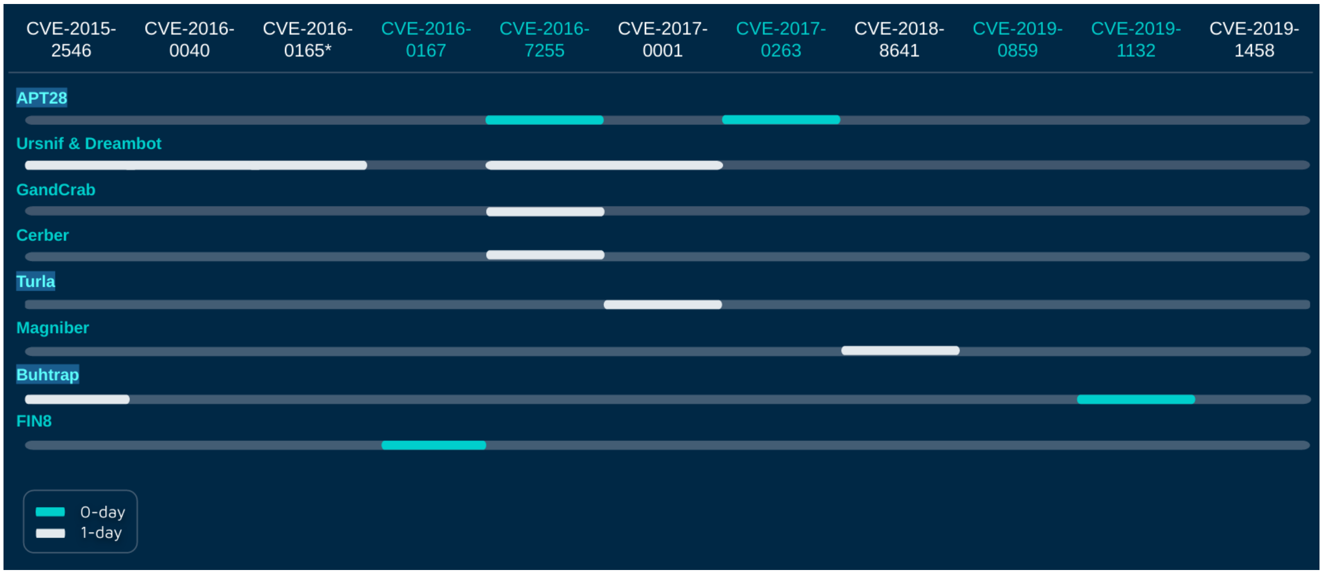
While there are additional aspects we could discuss such as different syscalls that each actor prefers to use during the exploitation process, naming conventions for created objects like Windows and ScrollBars, we believe that the list above clearly demonstrates the efficiency/validity of our approach. As can be seen from the list above, almost every aspect in an exploit can be implemented in several different ways. Still, both of our actors were very consistent in their respective exploitation routines, each sticking to their favorite way.

The Customers

During our entire research process, we wanted to focus on the exploit authors themselves, whether Volodya, PlayBit or others. And yet, we think that there is also much to learn by looking at these exploit authors' clientele. The list of Volodya's clients is diverse and includes banker trojan authors such as Ursnif, ransomware authors such as GandCrab, Cerber and Magniber, and APT groups such as Turla, APT28 and Buhtrap (which started from cyber-crime and later shifted to cyber-espionage). Interestingly, we can see that Volodya's 0-days are more likely to be sold to APT groups while 1-days are purchased by multiple crimeware groups. Without further intel, we can only assume that once a 0-day is detected by the security industry, the exploit is then recycled and sold at a lower price as a non-exclusive 1-day.

The APT customers, Turla, APT28, and Buhtrap, are all commonly attributed to Russia and it is interesting to find that even these advanced groups purchase exploits instead of developing them in-house. This is another point which further strengthens our hypothesis that the written exploits can be treated as a separate and distinct part of the malware.

The following table summarizes and shows the CVEs we were able to attribute to Volodya, as well as the customers or the malware groups we found using these exploits. CVEs that are marked with blue are 0-days, and naturally more expensive. The highlighted groups on the left are considered APTs.



	CVE-2015-2546	CVE-2016-0040	CVE-2016-0165*	CVE-2016-0167	CVE-2016-7255	CVE-2017-0001	CVE-2017-0263	CVE-2018-8641	CVE-2019-0859	CVE-2019-1132	CVE-2019-1458
APT28					0-day		0-day				
Ursnif & Dreambot	1-day	1-day	1-day		1-day	1-day					
GandCrab					1-day						
Cerber					1-day						
Turla						1-day					
Magniber								1-day			
Buhtrap	1-day									0-day	
FIN8				0-day							

Figure 11: Volodya’s customers and the CVEs that were used by them.

They grow up so fast

Before reviewing different trends we noted while examining the exploit samples over a period of time, we should emphasize that we have limited visibility as we can’t discuss 0-Days that weren’t caught yet. In addition, we can only attempt to date samples to the period before they were caught, but the sad truth is that we are usually pretty much bound to the date in which the exploit was actually first seen in the wild. Moreover, it is important for us to mention that it was clear from the start that Volodya was already quite professional when developing the first exploit we were able to attribute to them – CVE-2015-2546. For example, it had a unique Arbitrary-Write primitive that we couldn’t trace to any other exploit tutorial / exploit.

During the analysis of the exploits, as well as the analysis of dozens of malware samples we collected, we noticed an interesting shift. While the earlier Volodya exploits were sold as source code to be embedded in the malware, the later exploits were sold as an external utility that accepts a certain API. This change can suggest that Volodya is taking more precautions.

During the time between 2015 and 2019, we also noticed significant improvements in Volodya’s technical skills. As they got better and more experienced, Volodya started using more effective Arbitrary Read and Write primitives and they even fixed a bug in these primitives between

CVE-2015-2546 and CVE-2016-0165*. Moreover, the code of the exploits became more modular, as large functions were split into smaller sub-routines. Also, their technique to search and access specific offsets in various structs was also improved and in recent implementations it became more dynamic and safe, as it better-handled changes in minor versions of Windows.

Not only does this show the learning curve and development of our actor, but it also hints at their skills. The ability to find and reliably exploit Windows Kernel vulnerabilities is really not that straightforward. We can see in comparison that PlayBit was pretty much very active in this market between the years 2015-2018, and their focus was on selling exploits for 1-Day vulnerabilities, one of which was a 0-Day of Volodya (CVE-2016-7255).

Conclusion

Our research methodology was to fingerprint an exploit writer’s characteristics and later on use these properties as a unique hunting signature. We deployed this technique twice when tracking down Volodya’s exploits and those of PlayBit. Having these two successful test cases, we believe that this research methodology can be used to identify additional exploit writers. We recommend other researchers try our suggested technique and adopt it as an additional tool in their arsenal.

During this research, we focused on the exploits that are used by or embedded in different malware families, both in APT attacks and in commodity malware (especially ransomware). Although they are widespread, we often found detailed malware reports that neglected to mention that the malware at hand also uses an exploit for escalating its privilege.

The fact that we were able to use our technique, repeatedly, to track 16 Windows LPE exploits, written and sold by two different actors, was very surprising. Considering that 15 of them date to the timeframe of 2015-2019, it is plausible to assume that they constitute a significant share of the exploitation market, specifically for Windows LPE exploits.

Finally, it is impossible to tell the overall number of Windows kernel 0-day vulnerabilities that are being actively exploited in the wild. Nation-state actors are less likely to get caught and thus the infosec community does not have clear visibility to their ammo crate. That said, we can still get insights by looking at the exploits that were caught, while remembering this survivorship bias. Last year, Kaspersky reported a single actor who distributed an exploit framework that includes 3 more 0-Days. Adding up these numbers, we see that 8 out of 15 zero-day exploits, more than half of the “market-share”, are attributed to only two actors(!). This means that our research technique could potentially be used to track down many of the actors in the seen market, if not all of them.

Recommendation for Protection

Check Point Threat Emulation provides protection against this threat:

- Trojan.Wins.Generic.F
- Trojan.Wins.Generic.G

Appendix – IOC Table

Volodya

CVE-2015-2546: 3f6fe68981157bf3e267148ec4abf801a0983f4cea64d1aaf50fecc97ae590d3
CVE-2016-0040:
0ea43ba3e1907d1b5655a665b54ad5295a93bda660146cf7c8c302b74ab573e9
CVE-2016-0165*:
f1842080b38b3b990ba3ccc1d55ceedd901d423b6b8625633e1885f0dadee4c2
CVE-2016-0167:
6224efee6665118fe4b5bfbc0c4b1dbe611a43a4b385f61ae33b0a0af230da4e
CVE-2016-7255:
a785ad170a38280fc595dcc5af0842bd7cabc77b86deb510aa6ebb264bf2c092
CVE-2017-0001:
ed7532c77d2e5cf559a23a355e62d26c7a036f2c51b1dd669745a9a577f831a0
CVE-2017-0263:

f9dca02aa877ad36f05df1ebb16563c9dd07639a038b9840879be4499f840a10

CVE-2018-8641*:

0829f90a94aea5f7a56d6ebf0295e3d48b1dffcfefe91c7b2231a7108fe69c5e

CVE-2019-0859 – Initial 64bit sample:

895ab681351439ee4281690df21c4a47bdeb6691b9b828fdf8c8fed3f45202d8

CVE-2019-0859 – Matching 32bit sample:

eea10d513ae0c33248484105355a25f80dc9b4f1cfd9e735e447a6f7fd52b569

CVE-2019-1458:

8af2cf1a254b1dafe9e15027687b0315493877524c089403d3ffffa950389a30

PlayBit

CVE-2013-3660:

9f1a235eb38291cef296829be4b4d03618cd21e0b4f343f75a460c31a0ad62d3

CVE-2015-0057:

8869e0df9b5f4a894216c76aa5689686395c16296761716abece00a0b4234d87

CVE-2015-1701 (yes, it is the same sample as CVE-2015-0057)**:**

8869e0df9b5f4a894216c76aa5689686395c16296761716abece00a0b4234d87

CVE-2016-7255:

5c27e05b788ba3b997a70df674d410322c3fa5e97079a7bf3aec369a0d397164

CVE-2018-8453:

50da0183466a9852590de0d9e58bbe64f22ff8fc20a9ccc68ed0e50b367d7043