VB2020 localhost

30 September - 2 October, 2020 / vblocalhost.com

GRAPHOLOGY OF AN EXPLOIT HUNTING FOR EXPLOITS BY LOOKING FOR THE AUTHOR'S FINGERPRINTS

Itay Cohen & Eyal Itkin

Check Point Research, Israel

itayc@checkpoint.com
eyalit@checkpoint.com

ABSTRACT

Zero-days that are exploited in the wild always gain a lot of attention, and rightly so. But while malware authors usually get all the credit, exploit writers – the ones who work hard to find a vulnerability and develop a top-notch exploit – often remain out of the spotlight.

In recent months, our Vulnerability and Malware Research teams joined efforts to focus on the exploits inside 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'. To date, we have managed to track down more than 10 (!) of their *Windows* Kernel Local Privilege Escalation (LPE) exploits, many of which were 0-days at the time of development.

BACKGROUND

Our story begins, as all good stories do, with an incident response case. When analysing 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'. In the absence of any online resource with this implementation of CVE-2019-0859 [1], we realized that we were not looking at a publicly available PoC, but rather a real-world exploitation tool. This intrigued us, and prompted us to dig deeper.

Reverse-engineering the exploit was pretty straightforward. 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 as if the exploit and the malware itself had not been written by the same people. The code quality, lack of obfuscation, PDBs and timestamps all pointed to this conclusion.

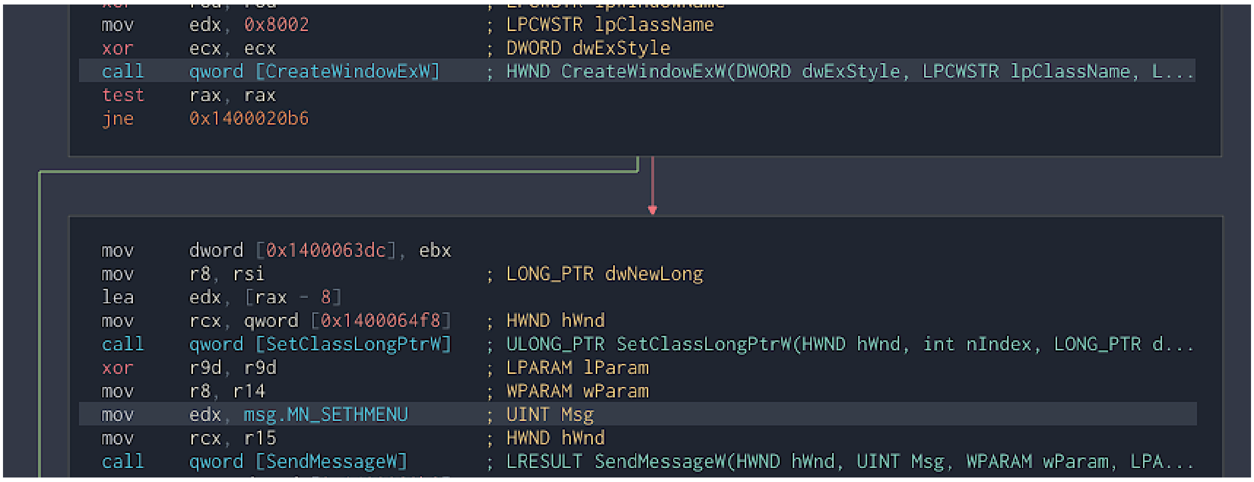

Figure 1: A call to CreateWindowEx, as seen in Cutter.

EXPLOIT DISTRIBUTION 101

We tend to look at the people behind a specific malware family as a single unit. It's easier to envision that each and every component was written by a single person, team, or group. The truth is that writing an advanced piece of malware, whether masterminded by a nation state or criminals, involves different groups of people with various skills. The 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 teams 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 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 the module and be done with it.

For this division of labour to work, the 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 exploits into their malware.

In essence, it is this integration point that is the key aspect that we will focus on in our research. Assuming that exploit authors work independently, and only distribute their code/binary modules to the malware authors, we decided to focus on them for a change. By analysing 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 to their identity when distributing their products to their malware-writing counterparts.

FINGERPRINTING EXPLOIT DEVELOPERS

Instead of focusing on a specific piece of malware and hunting for new samples of the malware family or actor, we wanted to offer another perspective and decided to concentrate on the few functions that were written by an exploit developer. Having the small 64-bit binary from our incident response case seemed 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 had been shared publicly. It made a great candidate for us to fingerprint, as the executable had been 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 piece of crimeware – which made us believe that the 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 paths 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', etc.)
    - 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 within it
        - Modularity: separation to functions
        - Structure: separation to clear phases (Init, config, spray, token swap, etc.)
        - 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.

Note: *A detailed list of such techniques can be found in the section named 'The author's fingerprints'.*

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 only had 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 another, dozens of samples started to appear, and with each one we improved our hunting rules and methodologies. Through careful analysis of the samples we were able to understand which samples exploited which CVE. Based on that, we created a timeline to understand whether the exploit was written as a 0-day before it was exposed, or whether it was 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: *Kaspersky* reported [2] some relevant information about them on several occasions [3], and *ESET* researchers mentioned some of Volodya's incriminating trails in their VB2019 talk [4] about Buhtrap.

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.

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 was 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* [5]

Found in the following malware samples: Ursnif, Buhtrap

Our exploit samples use a different memory shaping technique from 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 found 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* [6]

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 the previous vulnerability: `Win32k!xxxMNDestroyHandler`.

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

Figure 2: Debug string indicating the confusion around CVE-2016-0165, as 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* [7], a technical report by *Trend Micro* [8]

Found in the following malware samples: Attributed to APT28 (a.k.a. Fancy Bear, Sednit). Used later by Ursnif, Dreambot, GandCrab, Cerber.

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 pieces of circumstantial evidence that suggest that this 0-day was the one that was mentioned by BuggiCorp in the famous ad [9] 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* [10]).

CVE-2017-0263

Classification: 0-day

Basic description: Use after free in `win32k!xxxDestroyWindow`

0-day vendor report: *ESET* [11]

Found in the following malware samples: Attributed to APT28 (a.k.a. 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 1-days is usually harder than identifying 0-days. This time we couldn't find any sample that might hint as to the identity of 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* labelled this vulnerability CVE-2018-8641, but we can't know for sure.

CVE-2019-0859

Classification: 0-day

Basic description: Use after free in `CreateWindowEx`

0-day vendor report: *Kaspersky* [12]

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 the clear PDB string: 'X:\tools**0day\09-08-2018**\x64\Release\RunPS.pdb', which differed from 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* [13]

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 methods 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 for which we have not yet found a sample, so we can't be certain 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 [14], detailed report [15])

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 its 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 six days after *Kaspersky*'s initial report.

Vulnerabilities summary

Below 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**

**See note attached to entry in previous list of exploits.*

As this shows, there is a clear shift from mainly exploiting 1-day vulnerabilities to mostly selling 0-day exploits.

THE AUTHOR'S FINGERPRINTS

Now that we have 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 that is deployed for the different exploits, as the function flow of each exploit – and even the order of the different functions – is shared between most of the exploits.

In 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. Using 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 as we used to hunt Volodya's exploits, we managed to track down five *Windows* LPE 1-day exploits that were written by PlayBit, in addition to other tools that the author sold over the years. We started from a single sample, CVE-2018-8453, which is used by the REvil ransomware, and used PlayBit's 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 one of Volodya's 0-days
- CVE-2018-8453

Technically, PlayBit also sold exploits for CVE-2019-1069 (a SandboxEscaper [16] vulnerability) and CVE-2020-0787. However, we ignored these exploits as they aren't memory corruption vulnerabilities but vulnerabilities in different services, and as such have a different structure.

bool elevate(int target_pid)

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

```
bool elevate(int target_pid)
```

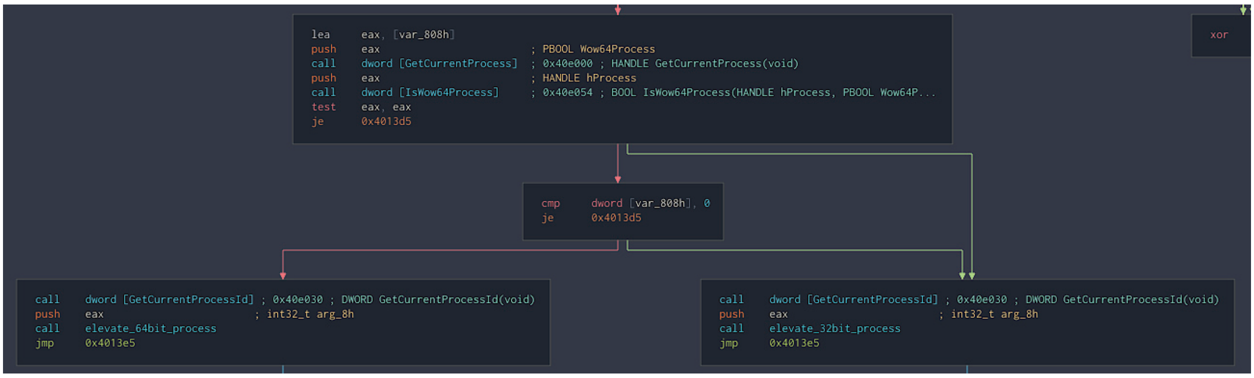

Figure 3: Invoking the elevate(target_pid) function, as seen in Cutter.

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 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 4: Starting the exploit with a call to Sleep(200), as seen in Cutter.

It is not entirely clear why the `Sleep(200)` is there in the template of the exploits. We suspect its purpose may be 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 prior to 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 nine of the exploits we've seen.

Comparison to PlayBit: There is no such feature in PlayBit's 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 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 [17].

```
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 5: Calling GetVersionExW() to get the Windows version, as seen in Cutter.

In both of these techniques, the goal is to query both the major and minor version of the OS, and to 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 or not it is a *Windows* server. Aside from an interest in specific *Windows 10* build versions, used only in the exploit for CVE-2019-1458, our actor uses only 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 6. 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 whether 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 6: Extracting the major and minor versions from the PEB, as seen in Cutter.

This is a clear difference in the modus operandi of the two 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.

We believe that this key difference is based on Volodya's capability to leak kernel addresses, a capability that PlayBit did not implement. More on that can be found in the next section.

Leaking kernel addresses

In the vast majority of exploits, the actor tunes the exploit and verifies the success of the memory shaping done to pool buffers/memory regions in kernel space using a leak primitive. In all exploits except CVE-2019-1458, this leak primitive is the well-known HMValidateHandle technique [18].

`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 objects in all *Windows* versions up to *Windows 10* RS4. This technique was well known and used even back in 2011, when 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 has continued to work quite well throughout the years, giving the actor no incentive to try to 'hide' by picking a lesser known function such as `CheckMenuRadioItem()`.

Comparison to PlayBit: There is no such feature in PlayBit's exploits.

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.

In the following subsections we outline 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 functions*

The *Windows* kernel implements the following functions for process-related functionality:

- `PsLookupProcessByProcessId` – retrieves a pointer to the process's `EPROCESS`.
- `PsInitialSystemProcess` – retrieves 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 two 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 code to be executed 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`
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.

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.

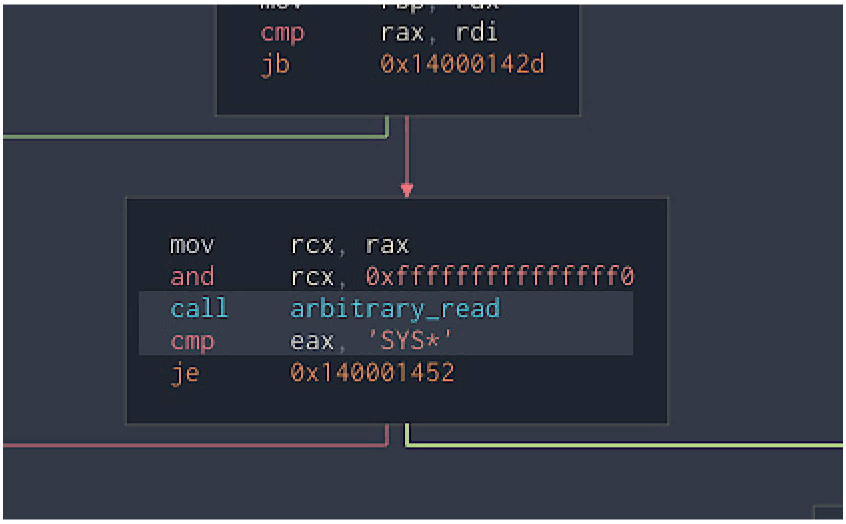

Figure 7: Volodya exploit using an Arbitrary-Read primitive in search for SYS, as seen in Cutter.*

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.

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

Comparison to PlayBit: In all of PlayBit's samples, we've always seen the use of Ps* functions for a token swap. This decision forced the actor to implement a few SMEP bypasses that were 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 analysed in *Microsoft*'s VB2015 paper [19]. Through this memory corruption, they can trigger a few memory leaks and memory 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 8: PlayBit exploit scanning the EPROCESS in search for the token, as 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 and 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 have been very consistent in their respective exploitation routines, each sticking to their favourite methods.

THE CUSTOMERS

During our 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 cybercrime 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.

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

THEY GROW UP SO FAST

Before reviewing the 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.

Having said that, the exploit samples that we saw are all concentrated in the time frame of 2015 to 2019. And if we re-examine our exploit table, we can easily identify a clear learning curve. While in 2015 our actor only exploited 1-days, in 2016 their first 0-day was already caught. In 2019 the actor already had three (caught) exploits, two of which were 0-days.

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

CONCLUSION

Our research methodology was to fingerprint an exploit writer's characteristics and 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 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 to 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 actively being exploited in the wild. Nation-state actors are less likely to get caught and thus the infosec community does not have clear visibility of their ammo crate. That said, we can still gain insights by looking at the exploits that have been caught, while remembering this survivorship bias. Last year, *Kaspersky* reported [20] a single actor who distributed an exploit framework that includes three more 0-days. Adding up these numbers, we see that eight 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.

REFERENCES

[1] CVE-2019-0859 | Win32k Elevation of Privilege Vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-0859.

[2] Cimpanu, C. Mysterious hacker has been selling Windows 0-days to APT groups for three years. ZDNet. May 2019. https://www.zdnet.com/article/mysterious-hacker-has-been-selling-windows-0-days-to-apt-groups-for-three-years/.

[3] Windows 0-day exploit CVE-2019-1458 used in Operation WizardOpium. SecureList. December 2019. https://securelist.com/windows-0-day-exploit-cve-2019-1458-used-in-operation-wizardopium/95432/.

[4] Buhtrap metamorphosis: from cybercrime to cyber espionage. https://www.youtube.com/watch?v=fsbLLAlC7sE.

[5] Two for One: Microsoft Office Encapsulated PostScript and Windows Privilege Escalation Zero-Days. FireEye. https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/twoforonefinal.pdf.

[6] Kizhakkinan, D.; Wang, Y.; Caselden, D.; Eng, E. Threat Actor Leverages Windows Zero-day Exploit in Payment Card Data Attacks. FireEye. May 2016. https://www.fireeye.com/blog/threat-research/2016/05/windows-zero-day-payment-cards.html.

[7] Mehta, N.; Leonard, B. Disclosing vulnerabilities to protect users. Google Security Blog. October 2016. https://security.googleblog.com/2016/10/disclosing-vulnerabilities-to-protect.html.

[8] Tang, J. One Bit To Rule A System: Analyzing CVE-2016-7255 Exploit In The Wild. Trend Micro Security Intelligence Blog. December 2016. https://blog.trendmicro.com/trendlabs-security-intelligence/one-bit-rule-system-analyzing-cve-2016-7255-exploit-wild/.

[9] Zero Day Auction for the Masses. SpiderLabs Blog. June 2016. https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/zero-day-auction-for-the-masses/.

[10] Jiang, G.; Lanstein, A.; Berry, A.; Read, B.; Kizhakkinan, D.; Macmanus, G. EPS Processing Zero-Days Exploited by Multiple Threat Actors. FireEye. May 2017. https://www.fireeye.com/blog/threat-research/2017/05/eps-processing-zero-days.html.

[11] Sednit adds two zero-day exploits using 'Trump's attack on Syria' as a decoy. We Live Security. May 2017. https://www.welivesecurity.com/2017/05/09/sednit-adds-two-zero-day-exploits-using-trumps-attack-syria-decoy/.

[12] Berdnikov, V.; Larin, B.; Ivanov, A. New zero-day vulnerability CVE-2019-0859 in win32k.sys. SecureList. April 2019. https://securelist.com/new-win32k-zero-day-cve-2019-0859/90435/.

[13] Cherepanov, A. Windows zero-day CVE-2019-1132 exploited in targeted attacks. We Live Security. July 2019. https://www.welivesecurity.com/2019/07/10/windows-zero-day-cve-2019-1132-exploit/.

[14] Windows 0-day exploit CVE-2019-1458 used in Operation WizardOpium. SecureList. December 2019. https://securelist.com/windows-0-day-exploit-cve-2019-1458-used-in-operation-wizardopium/95432/.

[15] Larin, B.; Kulaev, A. The zero-day exploits of Operation WizardOpium SecureList. May 2020. https://securelist.com/the-zero-day-exploits-of-operation-wizardopium/97086/.

[16] https://twitter.com/sandboxbear?lang=en.

[17] GetVersionExA function. https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getversionexa.

[18] A simple protection against HMValidateHandle technique. THEEVILBIT BLOG. July 2019. https://theevilbit.github.io/posts/a_simple_protection_against_hmvalidatehandle_technique/.

[19] Oh, J.W.; Florio, E. Duqu 2.0 Win32k Exploit Analysis. https://www.virusbulletin.com/uploads/pdf/conference_slides/2015/OhFlorio-VB2015.pdf.

[20] BlueHat IL 2020 – Boris Larin – Retrospective on the Latest Zero-days Found in the Wild. https://www.youtube.com/watch?v=UixFnBdIEzY.

APPENDIX: IOCs

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 64-bit sample: 895ab681351439ee4281690df21c4a47bdeb6691b9b828fdf8c8fed3f45202d8

CVE-2019-0859 – matching 32-bit sample: eea10d513ae0c33248484105355a25f80dc9b4f1cfd9e735e447a6f7fd52b569

CVE-2019-1458: 8af2cf1a254b1dafe9e15027687b0315493877524c089403d3ffffa950389a30

PlayBit

CVE-2013-3660: 9f1a235eb38291cef296829be4b4d03618cd21e0b4f343f75a460c31a0ad62d3

CVE-2015-0057: 8869e0df9b5f4a894216c76aa5689686395c16296761716abece00a0b4234d87

CVE-2015-1701: 8869e0df9b5f4a894216c76aa5689686395c16296761716abece00a0b4234d87 (yes, it is the same sample as CVE-2015-0057)

CVE-2016-7255: 5c27e05b788ba3b997a70df674d410322c3fa5e97079a7bf3aec369a0d397164

CVE-2018-8453: 50da0183466a9852590de0d9e58bbe64f22ff8fc20a9ccc68ed0e50b367d7043