Retrospective on the latest zero-days found in the wild

Boris Larin @oct0xor

kaspersky
Boris Larin
Senior Security Researcher at Kaspersky GReAT

- Likes reverse engineering (firmwares, kernels, etc.)
  - First person to participate in private Sony PlayStation BugBounty program 😊
- Finds zero-days exploited in the wild
- Finds supply chain attacks
  - ASUS “Operation ShadowHammer” and few others

Previously presented at:
Virus Bulletin, CanSecWest, SAS, BlueHat, TyphoonCon, ISC by Qihoo 360, AVAR, Code Blue, C3 ...

Twitter: @oct0xor
Previously on BlueHat ...

Overview of the latest Windows OS kernel exploits found in the wild

30-May-19
What this talk is about

We are ready to share the next chapter of our in the wild zero-day finding journey.

BlueHat Shanghai 2019:
- 2 LPE’s for the latest build of Windows 10 at the moment of discovery
- Sneak peek into exploitation framework that was used among many 0day exploits

BlueHat IL 2020:
- Detailed look into exploitation framework
  - Zero-days as a service
  - Hiding from AV vendors in the era of System Guard Runtime Attestation
- Information about some new LPE exploits that we found in 2019
- Information about full chain that we found in 2019: Chrome RCE + Windows EOP
BlueHat Shanghai 2019 rewind

In 2018 we caught 4 zero-days in the wild:

- **May 2018** – CVE-2018-8174 – Windows VBScript Engine RCE Vulnerability
  Attack on Microsoft Office through Internet Explorer

- **October 2018** – CVE-2018-8453 – Win32k EOP Vulnerability
  First known exploit targeting Win32k in Windows 10 RS4

- **November 2018** – CVE-2018-8589 – Win32k EOP Vulnerability
  Exploit targeted only Windows 7 SP1 32-bit but sophisticated framework is discovered

  Lets to escape sandboxes of web browsers, reveals novel exploitation techniques

Two last exploits were a part of the same exploitation framework
Zero-days as a service

Zero-day exploits is a multi-billion dollar industry

A few companies working in this area are well known and have a lot of publicity
Some others are not

We observe a rapid growth of entities who develop and sell exploits in volumes
Each with different business model

Some are picky to choose customers
There is a possibility that some are not, and those companies do greater harm
Zero-days used in the wild are too risky to be silent about if you encounter one
Exploitation framework

When products of such companies are used in the wild its just a matter of time until we or our industry partners stumble into one of them.

We have found attacks with the use of a unique exploitation framework.

Attacks are conducted by several threat actors.

Exploitation framework seemingly contained an armory of different exploits.

Functionality of framework also includes:

- AV evasion
- Choosing appropriate exploit reliably
- Sophisticated direct kernel object manipulation
I am often asked:

“How do you even find zero-days? Don't they check that AV is installed?”

Probably it would be perfect if this was the case

It would mean that if you install AV then you will not be attacked with zero-day (If attacker cares about not burning zero-day of course)

Right now all PCs have AV installed by default (e.g. Windows Defender)
Imagine a product which is basically a collection of zero-days
There are two sides: manufacturer and customer

Manufacturer
It makes sense to protect zero-days because it’s a valuable asset
They are shared with other and potentially new customers

Customer
Successes of operation is what matters the most
Depending on a license model customer might not care about wasting zero-day

One customer burns zero-day before others use it, everyone’s loss?

Seems that protection of zero-day is a problem of manufacturer, if sold non-exclusive
Exploitation framework – AV evasion

AV evasion starts from the moment when shellcode is executed as a part of full chain.

All API functions are executed using specially built trampolines or system calls. Framework searches for code patterns in text section of system libraries. Uses found gadgets to build fake stack and execute functions.

```c
/* build fake stack */
push ebp
mov ebp, esp
push offset gadget_ret
push ebp
mov ebp, esp
push offset gadget_ret
... 

/* push args */
...

/* push return address */
push offset trampoline_prolog

/* jump to function */
jmp eax
```
Exploitation framework – AV evasion

Shellcode is used to execute embedded PE module

This embedded module is a central part of framework
It contains logic to execute exploits and backdoor

All API functions are executed using the same trampoline technique as in shellcode

But embedded module also contains additional AV evasion logic

Implementation for this logic is quite strange
A dedicated function checks if special libraries are present in exploited process
List of libraries depends on version of operating system
In case if library is found in process exploits are not executed
Exploitation framework – AV evasion

Older builds

```c
switch ( windows_version_id )
{
    case WIN_10:
        libs = {"emet.dll", "emet64.dll"}; break;
    case WIN_8_1:
    case WIN_8:
    case WIN_7_SP1:
    case WIN_7:
    case WIN_XP:
        // BitDefender
        libs = {"avcuf32.dll", "avcuf64.dll"}; break;
    default:
        libs = {};
}
```

Newer builds

Same thing with small modifications:

Now library names are checked using CRC checksum over wide strings

For WIN_10 new libs added:
"avcuf32.dll", "avcuf64.dll", "mbae.dll", "mbae64.dll", +1 unknown library

In other cases:
"mbae.dll", "mbae64.dll", +1 unknown library

This check takes place before execution of EOP exploits, additional check happens after
RCE exploit may be triggered more than once
For reliable exploitation a proper mutual exclusion is required
Otherwise execution of multiple instances of EOP exploit will lead to BSOD
Use of `CreateMutex()` function may arouse suspicion
Framework uses quite interesting trick to implement custom mutex
HANDLE heap = GetProcessHeap();

HeapLock(heap);

while (HeapWalk(heap, &Entry))
{
    if (Entry.wFlags & PROCESS_HEAP_ENTRY_BUSY
     && Entry.cbData == size
     && memcmp(Entry.lpData, data, size))
    {
        return -1;
    }
}

HeapUnlock(heap);

void* buf = HeapAlloc(heap, HEAP_ZERO_MEMORY, size);
memcpy(buf, data, size);

Look for special memory block. If it exists, then exploit is running.

If memory block is not found, create it and we have a mutex.
Exploitation framework – Armory

Framework module may come with multiple exploits (embedded or received remotely). Each elevation of privilege exploit is implemented as a module with special interface. Exploits check version of operating system to find out if exploit supports target. Framework tries different exploits until it finds the one that succeeds.

```c
while (!found)
{
    get_exploit(&exploit)
    if (execute_exploit(exploit, ...))
    {
        found = 1;
    }
    if (++count >= 10)
        break;
}
```

Maximum number of EOP exploits is hardcoded to 10.
Exploitation framework – Armory

Each exploit contained a debug information that reveals codename of exploit

Developers granted each elevation of privilege exploit with a girl name

We have found 4 exploits:

- Alice – CVE-2018-8589
- Christine – CVE-2015-2360
- Dana – CVE-2019-0797
- Jasmine – CVE-2018-8611
Exploitation framework – Armory

Naming pattern:

- Alice - CVE-2018-8589
- ...  
- Christine - CVE-2015-2360
- Dana - CVE-2019-0797
- ...  
- Jasmine - CVE-2018-8611

A B C D E F G H I J

In English alphabet ‘J’ has an index 10

Jasmine also looks like the most modern exploit

```c
while ( !found )
{
    get_exploit(&exploit)

    if ( execute_exploit(exploit, ...) )
    {
        found = 1;
    }

    if ( ++count >= 10 )
        break;
}
```
Exploitation framework – Armory

We looked, but could not find any more exploits from this framework
But we shared this information with our partner
Using information that we provided, our partner was able to find more zero-days

Vulnerabilities were fixed 😊
We believe that the main purpose of this framework is a full chain attacks
We succeeded in analysis of elevation of privilege exploits and backdoor module
But our knowledge about remote code execution exploits is limited
However, we are aware that this framework was observed to be used as a payload for Adobe Flash zero-day CVE-2018-5002

Most recent discoveries

In 2019 we caught 4 zero-days in the wild:

- March 2019 - CVE-2019-0797 - Win32k EOP Vulnerability
- April 2019 - CVE-2019-0859 - Win32k EOP Vulnerability
- December 2019 - CVE-2019-1458 - Win32k EOP Vulnerability
Race condition in win32k.sys driver

Abused in “Dana” elevation of privilege exploit

The last 0day that was used by exploitation framework and discovered by us

Exploit code is written to support next OS versions:

- Windows 10 builds 10240, 10586, 14393, 15063
- Windows 8.1
- Windows 8
Win32k driver contains code for DirectComposition API

DirectComposition

• Introduced in Windows 8
• Lets to combine and animate elements
  • Bitmap composition with transforms, effects, and animations
  • Combine bitmaps of different sources (GDI, DirectX...)
  • Using tree like structure similar to Visual Tree in XAML
• Relatively new part of Win32k but already was exploited before

Look for `NtDComposition*` syscalls

dq offset NtDCompositionAddCrossDeviceVisualChild
dq offset NtDCompositionAddVisualChild
dq offset NtDCompositionBeginFrame
dq offset NtDCompositionCommitChannel
dq offset NtDCompositionConfirmFrame
dq offset NtDCompositionConnectPipe
dq offset NtDCompositionCreateAndBindSharedSection
dq offset NtDCompositionCreateChannel
dq offset NtDCompositionCreateConnection
dq offset NtDCompositionCreateDwmChannel
dq offset NtDCompositionCreateResource
dq offset NtDCompositionCurrentBatchId
dq offset NtDCompositionDestroyChannel
dq offset NtDCompositionDestroyConnection
dq offset NtDCompositionDiscardFrame
dq offset NtDCompositionDuplicateHandleToProcess
dq offset NtDCompositionDwmSyncFlush
dq offset NtDCompositionConnectPipe
dq offset NtDCompositionGetConnectionBatch
dq offset NtDCompositionGetDeletedResources
dq offset NtDCompositionGetFrameLegacyTokens
dq offset NtDCompositionGetFrameStatistics
dq offset NtDCompositionGetFrameSurfaceUpdates
dq offset NtDCompositionOpenSharedResource
NtDCOMpositionDiscardFrame

ExAcquirePushLockSharedEx((char *)connection + 0x88, 1164);
for ( i = (volatile signed __int32 *)((QWORD *)connection + 0x16)
i != (volatile signed __int32 *)((char *)connection + 0xA8);
i = (volatile signed __int32 *)((QWORD *)i + 1) )
{
    if ( *((QWORD *)i + 5) == FrameId )
    {
        _InterlockedIncrement(i - 2);
        Frame_ptr = (__int64)(i - 2);
        frame = (DirectComposition::CCompositionFrame *)(i - 2);
        status = 0;
        break;
    }
}
ExReleasePushLockSharedEx((char *)connection + 0x88, 1164);
v2h = status;
if ( (status >= 0 )
{
    ... 
    if ( !_InterlockedDecrement((volatile signed __int32 *)frame_ptr) )
    {
        *(DWORD *)(frame_ptr + 64) += 3;
        DirectComposition::CCompositionFrame::Discard(frame);
        Win32FreePool((void *)frame);
    }
}
DirectComposition::CConnection::RemoveCompositionFrame(connection, FrameId);
void __fastcall DirectComposition::CConnection::Disconnect(DirectComposition::CConnection *this)
{
    ...
    v1 = this;
    v2 = 0;
    DirectComposition::CCriticalSection::AcquireExclusive(*(DirectComposition::CCriticalSection **)((_QWORD *)this + 17) + 24164));
    DirectComposition::CCriticalSection::AcquireExclusive(*((DirectComposition::CCriticalSection **)v1 + 1));
    if ( *((_DWORD *)v1 + 0x21) )
    {
        *((_DWORD *)v1 + 0x21) = 0;
        v2 = 1;
    }
    DirectComposition::CConnection::DiscardAllCompositionFrames(v1);
    DirectComposition::CBatchSharedMemoryPoolSet::FreeAllPools((DirectComposition::CC
Execution of `NtDCompositionDiscardFrame` and `NtDCompositionDestroyConnection` simultaneously leads to a use-after-free.

`DiscardAllCompositionFrames` may be executed at a time when the `NtDCompositionDiscardFrame` syscall is looking for a frame to release or has already found it.
Exploit uses two distinct exploitation techniques

1) Abuse of GDI Palettes
   - Windows 10 builds 10240, 10586
   - Windows 8.1
   - Windows 8

2) Abuse of Windows
   - Windows 10 builds 14393, 15063
Abuse of GDI Palettes

1) Grooms heap using pallets, their addresses are leaked via GdiSharedHandleTable
2) Keeps a couple of pallets allocated next to each other
3) Uses vulnerability to free pallet #1, reclaim memory and overwrite cEntries field SetPaletteEntries on pallet #1 can be used to overwrite *pFirstColor of pallet #2
5) Arbitrary memory R/W using GetPaletteEntries / SetPaletteEntries

typedef struct _PALETTE
{
    ULONG cEntries;
    ...
    PALETTEENTRY *pFirstColor;
    ...
} PALETTE, *PPALETTE;

https://sensepost.com/blog/2017/abusing-gdi-objects-for-ring0-primitives-revolution/
https://github.com/sensepost/gdi-palettes-exp
Abuse of Windows

1) Grooms heap using `NtUserCreateInputContext` and `CreateWindowExW`
   - Window kernel address (win32k!tagWND) is leaked via `user32!gSharedInfo`

2) Uses vulnerability to free window, reclaim memory and overwrite `strName` field

3) Arbitrary memory R/W using `InternalGetWindowText` / `NtUserDefSetText`

```c
typedef struct tagWND
{
    ...
    LARGE_UNICODE_STRING strName;
    ...
} ...
```
Exploitation framework – Direct kernel object manipulation

In this framework each elevation of privilege exploit provides interfaces for DKOM

```java
class Kernel_IO {
    ReadQword(...)    WriteQword(...)
    ReadDword(...)    WriteDword(...)
    ReadWord(...)    WriteWord(...)
    ReadBytes(...)    WriteBytes(...)
    ...
    SetAddress(...)   ...
}
```

A special functionality for execution of kernel shellcode is also provided
Almost every EOP exploit that we observe use data-only exploitation

The most common scenario: walk EPROCESS structures and steal SYSTEM token
But this exploitation framework uses DKOM to get kernel code execution

A sophisticated functionality that is not commonly observed in EOP exploits

In that case it's worth taking a closer look on actual implementation and kernel shellcode

We also are lucky to see evolution of this functionality in framework
Exploitation framework – From R/W to code execution in kernel level

First is implementation observed among early variants

Step 1:
Shellcode is put into some kernel object whose kernel address is possible to leak
  - E.g. `CreateAcceleratorTableW + user32!gSharedInfo`

Step 2:
Duplicate handle to current thread, parse `EPROCESS->ObjectTable` to get address of `KTHREAD`
Get `KTHREAD->SchedulerApc`, verify it using type from `nt!KOBJECTS` and address of `KTHREAD`
Address of `KAPC.NormalRoutine` is stored for later use
Step 3:
Resolve base addresses of page table entries and page directory entries

- In Windows 10 build 10586 and below PT/PD entries are at fixed addresses
  - $\text{PT\_BASE} = 0x\text{FFFFFF680000000000}$
  - $\text{PD\_BASE} = 0x\text{FFFFFF6FB40000000}$
- In other cases base addresses are resolved heuristically from kernel image

Accessing page directory entry for VA:
- $\text{PDE\_ADDRESS} = \text{PD\_BASE} + ((\text{VA} \gg 18) \& 0x\text{FFFFFF8})$

Accessing page table entry for VA:
- $\text{PTE\_ADDRESS} = \text{PT\_BASE} + ((\text{VA} \gg 9) \& 0x\text{FFFFFF8})$
Exploitation framework – From R/W to code execution in kernel level

Modification of NX bit makes page executable
Owner bit is also interesting: it indicates kernel/user page
Exploitation framework – From R/W to code execution in kernel level

Step 4:
KAPC.NormalRoutine is overwritten with shellcode address
Current thread is suspended to get shellcode executed

Step 5:
KAPC.NormalRoutine original value is restored
Current thread is resumed
Exploitation framework – From R/W to code execution in kernel level

Kernel shellcode borrows a lot from open source Blackbone project

**Code is very similar to BlackBoneDrv/Inject.c**

Backdoor module is injected and executed inside chosen process (e.g. svchost.exe)

Injection is achieved exactly like in BlackBoneDrv:
- Reflective PE loading from kernel
- User code is executed using trampolines and creation of worker thread

---

Blackbone source code is known to be used in malware
Probably that's the reason why in newer variants this part was re-done

https://github.com/DarthTon/Blackbone
Exploitation framework – From R/W to code execution in kernel level

In newer variants the following technique is used

**Step 1:**
Dummy thread with `SignalObjectAndWait` is created and executed.
- `EPROCESS->ObjectTable` is parsed to get address of `KTHREAD`.

**Step 2:**
- Thread stack base and thread stack limit values are retrieved.
- Stack is parsed in search of address belonging to kernel image `.text` section.
- Bytes at correct return address should be equal to `8B D8 89 44`.

Exploitation framework – From R/W to code execution in kernel level

Step 3:
When correct address is found a ROP chain is build and written to the stack
Event to thread is fired to get **ROP chain executed**
In this case kernel shellcode is never used

This technique lets to execute any kernel function from user level!
Exploitation framework – From R/W to code execution in kernel level

Exploitation framework ➔ Kernel ➔ System process (e.g. svchost.exe)

Function call via ROP ➔ Code injection
Future of direct kernel object manipulation

DKOM lets to use many interesting techniques other than stealing SYSTEM token
Advanced threat actors already using them

Living in kernel land have a number of advantages
It’s the same level where AV products live
Possible attacks on System Guard Runtime Attestation

We are likely to see more malicious uses of direct kernel object manipulation

Other kernel exploits found in 2019

CVE-2019-0859 - Use-After-Free in win32k.sys driver

Root cause: Ability to set improper extra data with SetWindowLongPtr
ASLR bypass: HMValidateHandle
Exploitation primitive: Windows

CVE-2019-1458 - Arbitrary Pointer Dereference in win32k.sys driver

Root cause: Ability to set improper extra data with SetWindowLongPtr
ASLR bypass: CreateAcceleratorTableA + user32!gSharedInfo / GdiSharedHandleTable
Exploitation primitive: Bitmaps

State of Windows OS kernel exploitation

The vast majority of exploits found in the wild support only older builds of OS Windows 10 is commonly supported, but only older builds

Exploits for CVE-2018-8453 and CVE-2018-8611 are unique there
They exploited the latest builds at the moment of discovery
More information about them in our slides for BlueHat Shanghai 2019

Finding vulnerabilities is much more easier than developing novel techniques for them

https://github.com/oct0xor/presentations
Beyond kernel exploitation

The majority of 0day exploits that we found in the wild recently are kernel exploits.
We also look for RCE zero-days, but there are some complications.

Our technologies are aimed at detection and prevention of exploitation.

Exploit detection != zero-day finding
Additional analysis is always required
Lack of information

At first signs of exploitation there already might be nothing in memory
All scripts were just-in-time compiled and freed from memory
There are no interfaces available to kindly ask a browser for those scripts

We work on those problems and zero-days gets found
CVE-2019-13720

Google Chrome Use-After-Free in Audio

Originally exploit supported only Chrome version 76 and 77
But we were able to fix exploit for the upstream versions and prove that its 0day

We call those attacks “Operation WizardOpium”
No definitive link with any known threat actors
Weak code similarities with Lazarus attacks (high chance of false alarm)
The profile of attack is more similar to DarkHotel attacks
Waterhole-style injection on a Korean-language news portal

Multiple redirects and layers of obfuscation and encryption
Actual exploit is split into multiple RC4 encrypted chunks
Key for actual payload is appended to GIF image
CVE-2019-13720

Vulnerability actually is a race condition that results in Use-After-Free (UaF)

1) Exploit triggers UaF to leak pointer (AudioFloatArray)
   • Defeats ASLR and makes it possible to retrieve others useful pointers
2) Memory is sprayed with attempt to reuse freed buffer
3) Control over AudioFloatArray lets to achieve arbitrary R/W
4) Exploit comes with huge WebAssembly object with “dummy” logic
5) V8 compiles WASM bytecode to native code and puts in into RWX section
6) RWX section is overwritten with shellcode
7) Exploit uses FileReader technique to trigger execution of shellcode
   • Technique is very similar to the one used in Chrome 72 FileReader UaF exploit

Detailed write-up will be published at securelist.com
Just the tip of the iceberg: zero-days found in the wild in 2019

CVE-2019-7286
CVE-2019-1458
CVE-2019-0859
CVE-2019-0676
CVE-2019-0797
CVE-2019-3568
CVE-2019-1367
CVE-2019-1132
CVE-2019-7287
CVE-2019-5768
CVE-2019-0803
CVE-2019-1429
CVE-2019-2215
CVE-2019-0808
CVE-2019-0880
CVE-2019-11707
CVE-2019-11708
CVE-2019-18187
CVE-2019-13720

https://googleprojectzero.blogspot.com/p/0day.html
Conclusions

Thanks to Microsoft and Google for handling our findings very fast

Sharing detailed information about vulnerabilities is important

- Particular thanks to Microsoft Active Protection Program (MAPP) team
- Details about our findings were shared with partners, similar vulns were found

Sharing insights about threat actors brings great results

- More in the wild zero-days gets found

Right now making zero-days is hard, but not hard enough yet; huge money involved

Exploits will act more stealthy on platforms with good visibility

- Better AV checks / evasion
- Novel uses of DKOM to bypass System Guard Runtime Attestation
Thank you!

Boris Larin @oct0xor

kaspersky