Extreme Privilege Escalation on Windows 8/UEFI Systems

Corey Kallenberg
Xeno Kovah
John Butterworth
Sam Cornwell

@coreykal
@xenokovah
@jwbutterworth3
@ssc0rnwell
Introduction

- **Who we are:**
  - Trusted Computing and firmware security researchers at The MITRE Corporation

- **What MITRE is:**
  - A not-for-profit company that runs six US Government "Federally Funded Research & Development Centers" (FFRDCs) dedicated to working in the public interest
  - Technical lead for a number of standards and structured data exchange formats such as **CVE**, **CWE**, **OVAL**, **CAPEC**, **STIX**, **TAXII**, etc
  - The first .org, !(mil | .gov | .com | .edu | .net), on the ARPANET
Attack Model (1 of 2)

- We’ve gained administrator access on a victim Windows 8 machine
- But we are still constrained by the limits of Ring 3
As attackers we always want

- More Power
- More Persistence
- More Stealth
Typical Post-Exploitation Privilege Escalation

- Starting with x64 Windows vista, kernel drivers must be signed and contain an Authenticode certificate.
- In a typical post-exploitation privilege escalation, attacker wants to bypass the signed driver requirement to install a kernel level rootkit.
- Various methods to achieve this are possible, including:
  - Exploit existing kernel drivers
  - Install a legitimate (signed), but vulnerable, driver and exploit it
- This style of privilege escalation has been well explored by other researchers such as [6][7].
- There are other, more extreme, lands the attacker may wish to explore.
There are other more interesting post-exploitation options an attacker may consider:
- Bootkit the system
- Install SMM rootkit
- Install BIOS rootkit
Modern platforms contain protections against these more exotic post-exploitation privilege-escalations

- Bootkit the system (Prevented by Secure Boot)
- Install SMM rootkit (SMM is locked on modern systems)
- Install BIOS rootkit (SPI Flash protected by lockdown mechanisms)
This talk presents *extreme* privilege escalation

- Administrator userland process exploits the platform firmware (UEFI)
- Exploit achieved by means of a new API introduced in Windows 8
Extreme Privilege Escalation (2 of 2)

- Once the attacker has arbitrary code execution in the context of the platform firmware, he is able to:
  - Control other "rings" on the platform (SMM, Ring 0)
  - Persist beyond operating system re-installations
  - Permanently "brick" the victim computer
Target Of Attack

- Modern Windows 8 systems ship with UEFI firmware
- UEFI is designed to replace conventional BIOS and provides a well defined interface to the operating system
Windows 8 API

SetFirmwareEnvironmentVariable function

Sets the value of the specified firmware environment variable.

Syntax

```
C++

BOOL WINAPI SetFirmwareEnvironmentVariable(
   _In_  LPCTSTR lpName,
   _In_  LPCTSTR lpGuid,
   _In_  PVOID pBuffer,
   _In_  DWORD nSize
);
```

- Windows 8 has introduced an API that allows a privileged userland process to interface with a subset of the UEFI interface
EFI Variable Creation Flow

- Certain EFI variables can be created/modified/deleted by the operating system
  - For example, variables that control the boot order and platform language
- The firmware can also use EFI variables to communicate information to the operating system
 EFI Variable Consumption

- The UEFI variable interface is a conduit by which a less privileged entity (admin Ring 3) can produce data for a more complicated entity (the firmware) to consume.
- This is roughly similar to environment variable parsing attack surface on *nix systems.
We’ve already co-discovered[setupforfailure] (with Intel team) some vulnerabilities associated with EFI Variables that allowed bypassing secure boot and/or bricking the platform.
However, VU #758382 was leveraging a proprietary Independent BIOS Vendor (IBV) implementation mistake, it would be more interesting if we could find a variable vulnerability more generic to UEFI.
UEFI Vulnerability Proliferation

- If an attacker finds a vulnerability in the UEFI "reference implementation," its proliferation across IBVs and OEMs would potentially be wide spread.
  - More on how this theory works "in practice" later…
Auditing UEFI

- UEFI reference implementation is open source, making it easy to audit
- Let the games begin:
  - Svn checkout https://svn.code.sf.net/p/edk2/code/trunk/edk2/

http://tianocore.sourceforge.net/wiki/Welcome
Where to start?

- Always start with wherever there is attacker-controlled input
- We had good success last year exploiting Dell systems by passing an attacker-crafted fake BIOS update…
- So let's see if UEFI has some of the same issues
To begin the process of sending a Capsule update for processing, the operating system takes a firmware capsule and fragments it across the address space.
Capsule Processing Initiation

- Operating system creates an EFI variable that describes the location of the fragmented firmware capsule
- A "warm reset" then occurs to transition control back to the firmware
Capsule Coalescing

- The UEFI code "coalesces" the firmware capsule back into its original form.
Capsule Verification

- UEFI parses the envelope of the firmware capsule and verifies that it is signed by the OEM
Capsule Consumption

- Contents of the capsule are then consumed….
  - Flash contents to the SPI flash
  - Run malware detection independent of the operating system
  - Etc…
Opportunities For Vulnerabilities

- There are 3 main opportunities for memory corruption vulnerabilities in the firmware capsule processing code:
  1. The coalescing phase
  2. Parsing of the capsule envelope
  3. Parsing of unsigned content within the capsule

- Our audit of the UEFI capsule processing code yielded multiple vulnerabilities in the coalescing and envelope parsing code:
  - The first "BIOS reflash" exploit was presented by Wojtczuk and Tereshkin and involved reading the UEFI code which handled BMP processing and exploiting an unsigned splash screen image embedded in a firmware.
Coalescing Bug #1

Bug 1: Integer overflow in capsule size sanity check
  – Huge CapsuleSize may erroneously pass sanity check

```c
EFI_STATUS
EFIAPI
CapsuleDataCoalesce (
    IN EFI_PEI_SERVICES **PeiServices,
    IN EFI_PHYSICAL_ADDRESS *BlockListBuffer,
    IN OUT VOID **MemoryBase,
    IN OUT UINTN *MemorySize
)
{
...
//
// Get the size of our descriptors and the capsule size. GetCapsuleInfo()
// returns the number of descriptors that actually point to data, so add
// one for a terminator. Do that below.
//
GetCapsuleInfo (BlockList, &NumDescriptors, &CapsuleSize);
if (((CapsuleSize == 0) || (NumDescriptors == 0)) {
    return EFI_NOT_FOUND;
}
...
DescriptorsSize = NumDescriptors * sizeof (EFI_CAPSULE_BLOCK_DESCRIPTOR);
...
if (*MemorySize <= (CapsuleSize + DescriptorsSize)) { <= Bug 1
    return EFI_BUFFER_TOO_SMALL;
}
```

Edk2/MdeModulePkg/Universal/CapsulePei/Common/CapsuleCoalesce.c
Coalescing Bug #2

- **Bug 2: Integer overflow in fragment length summation**
  - CapsuleSize may be less than true summation of fragment lengths

```c
EFI_STATUS
GetCapsuleInfo (  
    IN EFI_CAPSULE_BLOCK_DESCRIPTOR   *Desc,
    IN OUT UINTN                     *NumDescriptors OPTIONAL,
    IN OUT UINTN                     *CapsuleSize OPTIONAL
)
{
    UINTN Count;
    UINTN Size;
    ...

    while (Desc->Union.ContinuationPointer != (EFI_PHYSICAL_ADDRESS) (UINTN) NULL) {
        if (Desc->Length == 0) {
            //
            // Descriptor points to another list of block descriptors somewhere
            //
            Desc = (EFI_CAPSULE_BLOCK_DESCRIPTOR *) (UINTN) Desc->Union.ContinuationPointer;
        } else {
            Size += (UINTN) Desc->Length; <= Bug 2
            Count++;
            Desc++;
        }
    }

    if (NumDescriptors != NULL) {
        *NumDescriptors = Count;
    }

    if (CapsuleSize != NULL) {
        *CapsuleSize = Size;
    }

Edk2/MdeModulePkg/Universal/CapsulePei/Common/CapsuleCoalesce.c
```
Envelope Parsing Bug (Bug #3)

- **Bug 3**: Integer overflow in multiplication before allocation
  - LbaCache may be unexpectedly small if NumBlocks is huge
Bug 4: Integer overflow in IsOverlapped

- Can erroneously return False if Buff1 + Size1 overflows
- This didn’t directly lead to a vulnerability but we had to abuse it to successfully exploit the other bugs
Vulnerabilities Summary

- We spent ~1 week looking at the UEFI reference implementation and discovered vulnerabilities in security critical code.
- The presence of easy to spot integer overflows in open source and security critical code is... disturbing.
  - Is no one else looking here?
Onward To Exploitation

- The aforementioned code runs with read-write-execute permissions
  - Flat protected mode with paging disabled
  - No mitigations whatsoever

- However, successful exploitation in this unusual environment was non-trivial

© 2014 The MITRE Corporation. All rights reserved.
Coalescing Exploit Attempt

- **Attempt #1**: Provide a huge capsule size and clobber our way across the address space to some function pointer on the stack area.
Coalescing Exploit Fail

- Overwriting certain regions of the address space had undesirable results
- We had to come up with an approach that skipped past the forbidden region

See whitepaper for full details on the exploitation technique
Coalescing Exploit Success

- Came up with a multistage approach that involved corrupting the descriptor array
  - Achieve surgical write-what-where primitive
  - Combined bugs #1, #2, #4 and abused a CopyMem optimization

See whitepaper for full details on the exploitation technique
Envelope Parsing Exploitation

- Exploitation of the "envelope parsing" bug was complicated for several reasons
- Note that in order to trigger the undersized LbaCache allocation, the NumBlocks value must be huge
  - This effectively means that the corrupting for() loops will never terminate

```c
FvbDev->LbaCache = AllocatePool (FvbDev->NumBlocks * sizeof(LBA_CACHE)); <= Bug 3
...
// Last, fill in the cache with the linear address of the blocks
//
BlockIndex = 0;
LinearOffset = 0;
for (PtrBlockMapEntry = FwVolHeader->BlockMap;
    PtrBlockMapEntry->NumBlocks != 0; PtrBlockMapEntry++) {
    for (BlockIndex2 = 0; BlockIndex2 < PtrBlockMapEntry->NumBlocks; BlockIndex2++) {
        FvbDev->LbaCache[BlockIndex].Base = LinearOffset;
        FvbDev->LbaCache[BlockIndex].Length = PtrBlockMapEntry->Length;
        LinearOffset += PtrBlockMapEntry->Length;
        BlockIndex++;
    }
}
```

Edk2/MdeModulePkg/Core/Dxe/FwVolBlock/FwVolBlock.c
Total Address Space Annihilation

- Loop will corrupt entire address space and hang the system
### Other Complications

- **LbaCache pointer is overwritten by the corruption, further complicating things**
- **Values being written during the corruption are not entirely attacker controller**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Attack Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DE1A944</td>
<td>Other FvbDev Members</td>
<td>???</td>
</tr>
<tr>
<td>3DE1A940</td>
<td>FvbDev-&gt;LbaCache (pointer)</td>
<td>AttackValue x 16</td>
</tr>
<tr>
<td>3DE1A910</td>
<td>FvbDev-&gt;Signature</td>
<td>AttackValue x 10</td>
</tr>
<tr>
<td>3DE1A89C</td>
<td>FvbDev-&gt;LbaCache[1].Length</td>
<td>AttackValue</td>
</tr>
<tr>
<td>3DE1A898</td>
<td>FvbDev-&gt;LbaCache[1].Base</td>
<td>AttackValue x 1</td>
</tr>
<tr>
<td>3DE1A894</td>
<td>FvbDev-&gt;LbaCache[0].Length</td>
<td>AttackValue</td>
</tr>
<tr>
<td>3DE1A890</td>
<td>FvbDev-&gt;LbaCache[0].Base</td>
<td>AttackValue x 0</td>
</tr>
</tbody>
</table>
Corruption Direction Change

- Overwriting the LbaCache pointer changes the location the corruption continues at.
Difficulties Recap

- We’ve got serious hoops to jump through to successfully exploit the envelope parsing vulnerability
  - Corrupting of base pointer for corruption (LbaCache)
  - Only partially controlled values being written
  - Corrupting loop will never terminate
Our approach to escaping the non-terminating for loop was to massage the corruption so the loop would self-overwrite.

In this case, we overwrite the top of the basic block with non-advantageous x86 instructions:

- Overwritten values only "semi-controlled"
- AttackerValue = 2D98BB8.
- Overwrites top of loop code on iteration=38E
- *(DWORD *)& 3EB21E44 = AttackerValue (B8 8B D9 02) [endianness]
Self-overwriting Success

With some brute force we discovered a way to overwrite the looping basic block with advantageous attacker instructions
- Jump to uncorrupted shellcode

- AttackerValue = 2D98CBF.
- Overwrites top of loop code on iteration=BB
- *(DWORD *)3EB21E42 = (AttackerValue * 0xBB) % 0x100000000
  = 14E9CF8F
  = 85 CF E9 14 [endianness]
- *(DWORD *)3EB21E46 = 8F 8C D9 02 [endianness]
Exploitation Mechanics Summary

- Vulnerable code runs with read-write-execute permissions and no mitigations
- However, successful exploitation was still complicated
- Capsule coalescing exploit allows for surgical write-what-where primitive resulting in reliable exploitation of the UEFI firmware
  - Address space is almost entirely uncorrupted so system remains stable
- Capsule envelope parsing vulnerability can be exploited but corrupts a lot of the address space
  - System probably in an unstable state
- In both cases, attacker ends up with control of EIP in the early boot environment
Our Sith attacker is unimpressed with his ring 3 admin privileges and seeks to grow his power.
Exploitation Flow (2 of 8)

- Attacker seeds an evil capsule update into memory
- Attacker then uses SetFirmwareEnvironmentVariable to prepare the firmware to consume the evil capsule
- Shellcode to be executed in the early boot environment is staged in memory
Exploitation Flow (3 of 8)

- Warm reset is performed to transfer context back to UEFI
Exploitation Flow (4 of 8)

- Capsule processing is initiated by the existence of the "CapsuleUpdateData" UEFI variable
Exploitation Flow (5 of 8)

- UEFI begins to coalesce the evil capsule

```
"CapsuleUpdateData" = 3E700000
```

```
UEFI Code
```

```
Evil DescriptorArray
```

```
Power Gaining Shellcode
```
Exploitation Flow (6 of 8)

- UEFI becomes corrupted while parsing evil capsule

```
"CapsuleUpdateData" = 3E700000
```

© 2014 The MITRE Corporation. All rights reserved.
Exploitation Flow (7 of 8)

- Attacker gains arbitrary code execution in the context of the early boot environment
  - Platform is unlocked at this point
Attacker can now establish agents in SMM and/or the platform firmware to do their bidding.
Unnatural Powers

- With these new powers, our attacker can:
  - Brick the platform
  - Defeat Secure Boot[2]
  - Establish an undetectable SMM rootkit[8][5]
  - Subvert hypervisors[9]
  - Subvert TXT launched hypervisors[3]
  - Circumvent operating system security functions[11]
  - Survive operating system reinstallation attempts
  - Other?
Demo Time
Vendor Response

- We told Intel & CERT about the bugs we found on Nov 22\textsuperscript{nd} and Dec 4\textsuperscript{th} 2013
  - We conveyed that we would extend our typical 6 month responsible disclosure deadline, and we would be targeting public disclosure in the summer at BlackHat/Defcon
  - We also directly contacted some of the OEMs that we already had the capability to send encrypted email to
- Intel queried UEFI partners to ask if they were using the affected code
- If the vendors said they thought they would be affected, then Intel sent them the details
- Then we didn't hear anything for a while
- Eventually Intel indicated which vendors said they were vulnerable, and which systems would be patched.
- This information is conveyed in CERT VU #552286
- The UEFI forum is in the process of setting up a UEFI Security Response Team to better coordinate these sort of disclosures in the future. Shooting to go live by Sept 1.
What can you do about it?

- Run Copernicus. It has been updated to automatically report if your system is on the VU # 552286 affected list
  - [http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about](http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about) or just search for "MITRE Copernicus"

- We also have a binary integrity checking capability for Copernicus. This can help you detect if your BIOS has been backdoored
  - The capability is freely available, but it's not as simple and foolproof as the public Copernicus (it will have false positives/negatives). And we don't really have the resources to support it for everyone. Therefore we prioritize who we work with to use it, based on the number of systems that will be checked. So if you're serious about checking your BIOSes, email copernicus@mitre.org

- We also need this data to feed further research results on the state of BIOS security in the wild on deployed systems. Unlike the IPMI people, we can't just portscan networks to get 100k research results :P
What can you do about it?

- If you're a security vendor, start including BIOS checks
  - If you're a customer, start asking for BIOS checks

- We are happy to freely give away our Copernicus code to get vendors started with incorporating checking BIOSes. All we ask for in return is some data to help further our research.

- We want BIOS configuration & integrity checking to become standard capabilities which are widely available from as many vendors as possible.
  - No more massive blind spot please!
Conclusion

- UEFI has more tightly coupled the bonds of the operating system and the platform firmware.

- Specifically, the EFI variable interface acts as a conduit by which a less privileged entity (the operating system) can pass information for consumption by a more privileged entity (the platform firmware).
  - We have demonstrated how a vulnerability in this interface can allow an attacker to gain control of the firmware.

- Although the authors believe UEFI to ultimately be a good thing for the state of platform security, a more thorough audit of the UEFI code and its new features is needed.

- Copernicus continues to be updated to give the latest information about whether vulnerabilities affect your BIOS.
Questions & Contact

- {ckallenberg, xkovah, jbutte<wbr/>rworth, scornwell}@mitre.<wbr/>org
- Copernicus@mitre.<wbr/>org
- @coreykal, @xenokovah, @jwbutterworth3, @ssc0rnwell
- @MITREcorp

- P.s., go check out OpenSecurityTraining.info!
- @OpenSecTraining
References

  http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
  http://blackhat.com/us-13/briefings.html#Bulygin
- [5] BIOS Chronomancy: Fixing the Core Root of Trust for Measurement – Butterworth et al., May 2013
  http://invisiblethingslab.com/resources/bh07/IsGameOver.pdf
  http://j00ru.vexillium.org/?p=1455
References 2