# Extreme Privilege Escalation On Windows 8/UEFI Systems

Corey Kallenberg Xeno Kovah John Butterworth
Sam Cornwell
ckallenberg@mitre.org, xkovah@mitre.org
jbutterworth@mitre.org, scornwell@mitre.org
The MITRE Corporation
Approved for Public Release
Distribution Unlimited. Case Number 14-2221

**Abstract**

The UEFI specification has more tightly coupled the bonds of the operating system and the platform firmware by providing the well-defined "Runtime Service" interface between the operating system and the firmware. This interface is more expansive than the interface that existed in the days of conventional BIOS, which has inadvertently increased the attack surface against the platform firmware. Furthermore, Windows 8 has introduced an API that allows accessing this UEFI interface from a privileged userland process. Vulnerabilities in this interface can potentially allow a privileged userland process to escalate its privileges from ring 3 all the way up to that of the platform firmware, which attains permanent control of the very-powerful System Management Mode. This paper discusses two such vulnerabilities that the authors discovered in the UEFI open source reference implementation and the techniques that were used to exploit them.

# Contents

# 1 Introduction

UEFI is rapidly replacing conventional BIOS on modern computers. A driving factor behind this migration is Microsoft's addition of UEFI firmware to the recommended hardware for Windows 8[1]. An important reason for Microsoft's push for UEFI adoption is the additional security features that UEFI provides. UEFI Secure Boot is one of these features which protects against bootkit style attacks that can compromise the integrity of the NT kernel at load time. Starting with Windows Vista, 64 bit editions of Windows have also enforced the requirement that kernel drivers be signed with an authenticode certificate. Thus the signed driver requirement coupled with Secure Boot enforces of the integrity of the ring 0 code in the Windows 8 x64 environment.

In the post exploitation phase, it may be desirable for an attacker to inject a rootkit into ring 0 in order to have powerful influence over the system. Due to Secure Boot and the signed driver requirement, the attacker would now require a ring 3 to ring 0 privilege escalation exploit that attacks a vulnerability in the NT kernel or a 3rd party driver. This particular attack model has already been discussed at length[12][13][14]. This paper instead seeks to explore a different method of post exploitation privilege escalation that allows the attacker permanent residence in an even more *extreme* environment... System Management Mode (SMM).

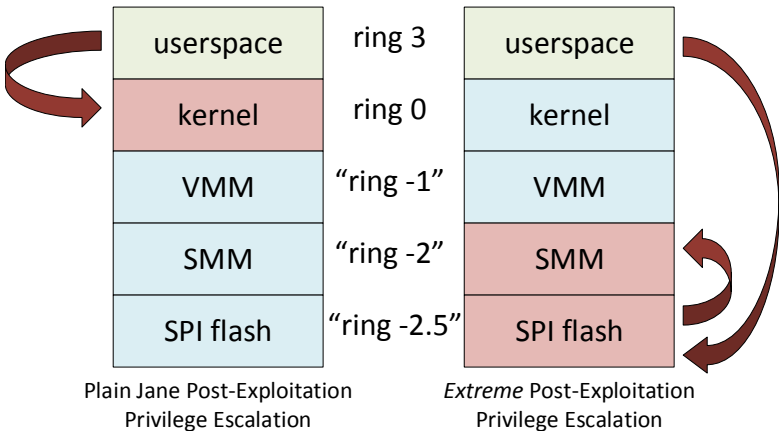


**Figure 1:** Plain Jane Post-Exploitation Privilege Escalation vs. *Extreme* Post-Exploitation Privilege Escalation

The attack surface explored in this paper is the UEFI Runtime Services interface. A successful attack against this interface may allow an attacker to permanently alter the UEFI firmware. From the UEFI firmware, the attacker is allowed to control the early bootup process of the system, including the configuration and initialization of the SMM code. This paper highlights the UEFI Runtime Services as a new and viable attack surface by describing and exploiting two UEFI vulnerabilities discovered by the authors.

# 2 Runtime Services

UEFI provides a set of functions that are accessible to both the early boot environment and to the operating system[18]. These functions are known as the "Runtime Services." The Runtime Services provide functionality to reset the system, modify firmware environment variables, initiate a firmware update, as well as other tasks. Typically these services are meant to be used by the operating system kernel. However, Windows 8 has introduced an API that exposes a subset of the Runtime Services to administrator userland processes.

[1] `http://windows.microsoft.com/en-us/windows-8/system-requirements`

## 2.1 Variable Interface

The Runtime Services provide functions for accessing "EFI Variables." EFI variables are similar to operating system environment variables. Typically EFI variables are consumed by the platform firmware during the boot up of the system. Alternatively, some EFI variables may be created by the firmware to communicate information to the operating system. For instance, the platform language and the boot media order are stored as EFI variables. The Runtime Services provide functions for reading, writing, creating and enumerating EFI variables. Furthermore, Windows 8 introduced the SetFirmwareEnvironmentVariable and GetFirmwareEnvironmentVariable functions for programmatically interacting with EFI variables from userland[11]. These functions are callable from an administrator userland process with the SE_SYSTEM_ENVIRONMNENT_NAME access token.

The important observation is the EFI variable interface is a conduit by which a less privileged domain (ring 3) can insert data for a more privileged domain (the platform firmware) to consume. Furthermore, many of these variables serve undocumented purposes and have complex contents. Historically this is the type of interface where memory corruption vulnerabilities have been discovered. Alert readers may draw comparisons to Unix environment variable parsing vulnerabilities[2]. In fact, vulnerabilities have already been discovered in some of these EFI variables that allowed bypassing Secure Boot or bricking the victim computer[16][4]. However, the aforementioned vulnerabilities were design flaws resulting from security critical configuration data being stored in an unprotected[3] EFI variable. This paper specifically considers memory corruption vulnerabilities that were found in the Intel UEFI reference implementation's[9] parsing of a standard EFI variable, "CapsuleUpdateData."

## 2.2 Capsule Update

The platform firmware is stored on a SPI flash chip that is soldered onto the motherboard. Because the firmware is a security critical component, Intel provides a number of chipset[5] flash protection mechanisms that can protect the contents of the flash chip from even ring 0 code. It is also necessary to implement a means to securely update the platform firmware in the event that bugs need to be patched, or new features added. Historically, the firmware update process was non standardized and OEM specific. UEFI attempts to standardize the firmware update process by defining "capsule update" functionality as part of the Runtime Services.

The capsule update Runtime Service seeds a firmware update capsule into RAM and then performs a soft reset of the system. During a warm reset of the system, the contents of RAM will remain intact, thus allowing the capsule contents to survive for consumption by the firmware. The flash chip is also unlocked as part of the reset. Early in the boot up of the system, the firmware will check for the existence of a firmware update capsule. If one exists, the firmware will verify the update contents are signed by the OEM, and if so, write the new firmware update to the still unlocked flash. If the update contents can not be cryptographically verified, or if no update is pending, the firmware locks the flash protection registers on the chipset to prevent further write access to the firmware. For further information on these flash protection mechanisms, the reader is referred to another paper[15][16].

Because an open source UEFI reference implementation is provided by Intel[8], the exact details of the UEFI capsule update implementation can be examined at the source code level. The implementation specifics are now described in detail.

### 2.2.1 Capsule Update Initiation

The capsule update process is initiated by calling the UpdateCapsule Runtime Service function.

[2]This class of vulnerability allowed an unprivileged user to escalate their privileges to root by seeding an environment variable with an exploit payload, and then calling a suid root program that unsafely parsed the relevant environment variable

[3]Non Authenticated, Runtime Accessible.

```
typedef
EFI_STATUS
UpdateCapsule (
  IN EFI_CAPSULE_HEADER   **CapsuleHeaderArray,
  IN UINTN                CapsuleCount,
  IN EFI_PHYSICAL_ADDRESS ScatterGatherList OPTIONAL
  );
```

**Listing 1:** UpdateCapsule definition.

The ScatterGatherList in Listing 1 is an array of EFI_CAPSULE_BLOCK_DESCRIPTOR entries. Each descriptor entry is a pair consisting of a capsule fragment data pointer, and a capsule fragment size.

```
typedef struct (
  UINT64 Length;
  union {
    EFI_PHYSICAL_ADDRESS DataBlock;
    EFI_PHYSICAL_ADDRESS ContinuationPointer;
  }Union;
) EFI_CAPSULE_BLOCK_DESCRIPTOR;
```

**Listing 2:** EFI_CAPSULE_BLOCK_DESCRIPTOR definition.

It is the responsibility of the calling operating system to decide how to fragment the contiguous update capsule so that it fits within the resource constraints of the system. Note that each individual fragment of the update capsule is unsigned. The location of the ScatterGatherList is stored in an EFI Non-Volatile variable named "CapsuleUpdateData" so that it can be passed onto the firmware during reboot. At this point, a warm reset is performed.

### 2.2.2 PEI Phase Capsule Coalescing

The UEFI boot process is divided into several phases. The Pre-EFI Initialization (PEI) phase occurs early in the boot up process and is responsible for, among other things, initializing the CPUs and main memory[7]. PEI is where the processing of the incoming capsule update image begins. Initially, an attempt is made to determine whether or not a firmware update is pending. If the platform is booting under a warm reset and the CapsuleUpdateData variable exists, the boot mode is changed to BOOT_ON_FLASH_UPDATE. At this point the contents of the CapsuleUpdateData variable is interpretted as a physical address pointing to the aforementioned ScatterGatherList.

Before processing can continue, the capsule update must be coalesced into its original form. The results of this process are described visually in Figure 2. After the update has been coalesced, further processing is deferred to the DXE phase.

### 2.2.3 DXE Phase Capsule Processing

The Driver Execution Environment Phase (DXE) is responsible for the majority of system initialization[6]. DXE is responsible for continuing to process the capsule image that was coalesced during PEI. The contents of the capsule image are encapsulated in a series of envelopes that provide contextual information about the contents of the update. For a visual depiction see Figure 3.

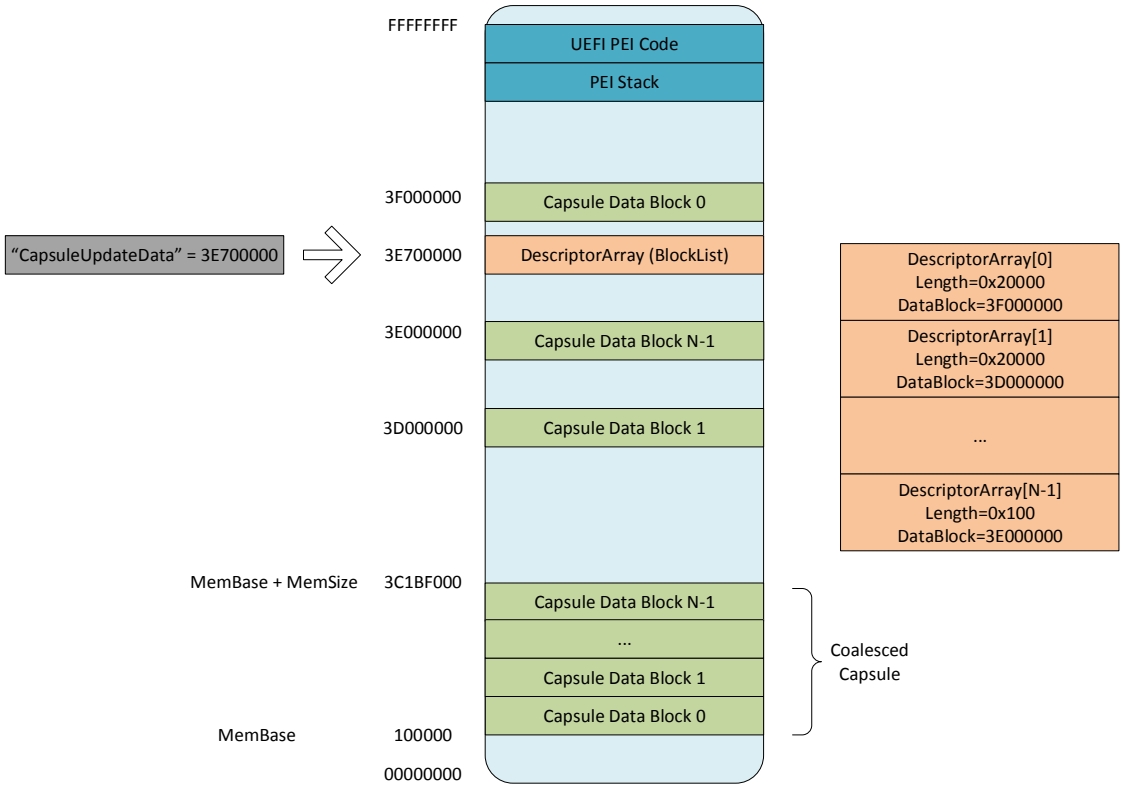


**Figure 2:** Capsule Image Coalesced During PEI Phase

```
typedef struct {
  EFI_GUID  CapsuleGuid;
  UINT32    HeaderSize;
  UINT32    Flags;
  UINT32    CapsuleImageSize;
} EFI_CAPSULE_HEADER;

typedef struct {
  UINT8                   ZeroVector[16];
  EFI_GUID                FileSystemGuid;
  UINT64                  FvLength;
  UINT32                  Signature;
  EFI_FVB_ATTRIBUTES      Attributes;
  UINT16                  HeaderLength;
  UINT16                  Checksum;
  UINT8                   Reserved[3];
  UINT8                   Revision;
  EFI_FV_BLOCK_MAP_ENTRY  FvBlockMap[1];
} EFI_FIRMWARE_VOLUME_HEADER;
```

**Listing 3:** Capsule Update envelope structures.

# 3 Capsule Update Vulnerabilities

The authors performed a brief 2 week audit of the open source UEFI reference implementation at release UDK2010[9]. The focus of the audit was the capsule update process, and the scope was limited to code that executes before cryptographic verification of the capsule contents. Critical vulnerabilities were found both

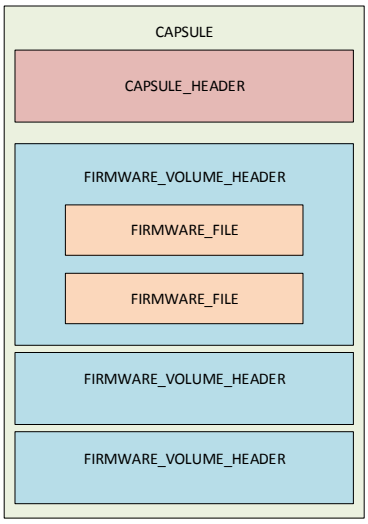


**Figure 3:** Capsule Envelopes

in the PEI coalescing phase, and in the DXE capsule processing phase. The specifics of the vulnerabilities are discussed below.

## 3.1 Coalescing Vulnerability

```
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;
  }
```

**Listing 4:** CapsuleDataCoalesce code

```
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;
  }
```

**Listing 5:** GetCapsuleInfo code

The important values for our discussion are CapsuleSize and DescriptorSize. CapsuleSize is set by GetCapsuleInfo and is equal to the sum of the length values in the descriptor array. DescriptorSize is also set by GetCapsuleInfo and is equal to the total size of the descriptor array. All of these values are attacker controlled.

There are several opportunities for integer overflow in the coalescing code descibed by Listings 4 and 5. The first bug is an integer overflow in the check to see if the CapsuleSize and DescriptorSize sum exceed the available MemorySize (Bug 1). The consequence of this overflow could be a very large CapsuleSize passing the buggy sanity check.

Another issue is an integer overflow possibility in the summation of the descriptor array Length members in GetCapsuleInfo (Bug 2). With this issue, if one entry in the descriptor array has a very large Length value, CapsuleSize could be less than the real sum of the Length values.

## 3.2 Envelope Vulnerability

```
typedef struct {
  UINTN                       Base;
  UINTN                       Length;
} LBA_CACHE;

typedef struct {
  UINT32 NumBlocks;
  UINT32 Length;
} EFI_FV_BLOCK_MAP_ENTRY;

typedef struct {
  UINTN                                 Signature;
  EFI_HANDLE                            Handle;
  EFI_DEVICE_PATH_PROTOCOL              *DevicePath;
  EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL    FwVolBlockInstance;
  UINTN                                 NumBlocks;
  LBA_CACHE                             *LbaCache;
  UINT32                                FvbAttributes;
  EFI_PHYSICAL_ADDRESS                  BaseAddress;
  UINT32                                AuthenticationStatus;
} EFI_FW_VOL_BLOCK_DEVICE;

EFI_STATUS
ProduceFVBProtocolOnBuffer (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length,
  IN EFI_HANDLE             ParentHandle,
  IN UINT32                 AuthenticationStatus,
  OUT EFI_HANDLE            *FvProtocol  OPTIONAL
  )
{
  EFI_STATUS                    Status;
  EFI_FW_VOL_BLOCK_DEVICE       *FvbDev;
  EFI_FIRMWARE_VOLUME_HEADER    *FwVolHeader;
  UINTN                         BlockIndex;
  UINTN                         BlockIndex2;
  UINTN                         LinearOffset;
  UINT32                        FvAlignment;
  EFI_FV_BLOCK_MAP_ENTRY        *PtrBlockMapEntry;

  FwVolHeader = (EFI_FIRMWARE_VOLUME_HEADER *)(UINTN) BaseAddress;
...
  //
  // Init the block caching fields of the device
  // First, count the number of blocks
  //
  FvbDev->NumBlocks = 0;
  for (PtrBlockMapEntry = FwVolHeader->BlockMap;
       PtrBlockMapEntry->NumBlocks != 0;
       PtrBlockMapEntry++) {
    FvbDev->NumBlocks += PtrBlockMapEntry->NumBlocks;
  }
  //
  // Second, allocate the cache
  //
  FvbDev->LbaCache = AllocatePool (FvbDev->NumBlocks * sizeof (LBA_CACHE)); <= Bug 3
```

**Listing 6:** ProduceFVBProtocolOnBuffer Code and Structures

The code in Listing 6 is called during the DXE phase to prepare the capsule for further processing. The NumBlocks member of FvbDev is set equal to the the summation of an attacker controlled array of UINT32 values that dangle off of the EFI_FIRMWARE_VOLUME_HEADER envelope. This array is of arbitrary

length, as summation is only terminated when a zero entry is encountered.

In the case where DXE executes in 32 bit mode, the multiplication involved in the allocation is trivially overflowable. In the case where DXE executes in 64 bit mode, FvbDev's NumBlocks is a 64 bit integer, but the attacker controlled entries in the BlockMap array retain 32 bit width. Hence to trigger this integer overflow, an attacker must cause FvbDev's NumBlocks to be sufficiently large through a series of 32 bit integer additions. An attacker who creates a BlockMap array with over 0x10000000 entries of maximum NumBlocks (0xffffffff) can force this integer overflow in the multiplication. This huge BlockMap array requires approximately 2GB of RAM to create, which is within the realm of possibility on modern systems where 4GB or more is often standard. The result of an overflow in the multiplication before the allocation will be an unexpectedly small LbaCache buffer (Bug 3).

# 4 Capsule Update Exploitation

Exploitation of these vulnerabilities proved to be sufficiently interesting to warrant discussion. The execution environment of the vulnerable code is atypical. The processor is running in protected mode with a flat segmentation model and paging disabled. Because memory protections are generally provided at the page level, with paging disabled, the majority of the address space is RWX with a few exceptions[4]. Also noteworthy is the complete lack of exploit mitigations in the firmware code and its execution environment.

Despite these attacker advantages, significant hurdles had to be overcome to successfully exploit the vulnerabilities. The primary obstacle for an attacker working in this space is the lack of appropriate debugging capabilities. To overcome this, the authors originally developed their exploits against the MinnowBoard[2]. The MinnowBoard was chosen because of its UEFI firmware (which contained all of the relevant vulnerabilities) and its provided debug stub. The following section discusses the exploitation process as it unfolded against the MinnowBoard. Exploitation of OEM specific code is discussed later in Section 7.2.

## 4.1 Coalescing Exploitation

During the capsule coalescing in the PEI phase, the processor is executing in 32 bit protected mode with paging disabled. The PEI code is executing in place out of flash memory and Cache-As-RAM (CAR) is being used for stack space. The MinnowBoard has 1GB of RAM, meaning that addresses between [0,0x3FFFFFFF] are backed up with physical frames. The CAR stack and PEI code are at the top of the address space. This means that a large gap exists in the middle of the address space that is neither backed up by devices nor RAM.

After the integers overflows described in Section 3 allow for an unexpectedly large capsule to be coalesced, the following code is relevant.

[4]Code executing in place out of flash memory is not writable.

```
  if (*MemorySize <= (CapsuleSize + DescriptorsSize)) {
    return EFI_BUFFER_TOO_SMALL;
  }

  FreeMemBase = *MemoryBase;
  FreeMemSize = *MemorySize;
...

  //
  // Take the top of memory for the capsule. Naturally align.
  //
  DestPtr         = FreeMemBase + FreeMemSize - CapsuleSize;
  DestPtr         = (UINT8 *) ((UINTN) DestPtr &~ (UINTN) (sizeof (UINTN) - 1));
  FreeMemBase     = (UINT8 *) BlockList + DescriptorsSize;
  FreeMemSize     = (UINTN) DestPtr - (UINTN) FreeMemBase;
  NewCapsuleBase  = (VOID *) DestPtr;

  //
  // Move all the blocks to the top (high) of memory.
  // Relocate all the obstructing blocks. Note that the block descriptors
  // were coalesced when they were relocated, so we can just ++ the pointer.
  //
  CurrentBlockDesc = BlockList;
  while ((CurrentBlockDesc->Length != 0) || (CurrentBlockDesc->Union.ContinuationPointer != (EFI_PHYSICAL_ADDRESS) (UINTN) NULL)
    //
    // See if any of the remaining capsule blocks are in the way
    //
    TempBlockDesc = CurrentBlockDesc;
    while (TempBlockDesc->Length != 0) {
      //
      // Is this block in the way of where we want to copy the current descriptor to?
      //
      if (IsOverlapped (
            (UINT8 *) DestPtr,
            (UINTN) CurrentBlockDesc->Length,
            (UINT8 *) (UINTN) TempBlockDesc->Union.DataBlock,
            (UINTN) TempBlockDesc->Length
            )) {
                  //Relocate the block
                  RelocPtr = FindFreeMem (BlockList, FreeMemBase, FreeMemSize, (UINTN) TempBlockDesc->Length);
                  ....
                  CopyMem ((VOID *) RelocPtr, (VOID *) (UINTN) TempBlockDesc->Union.DataBlock, (UINTN) TempBlockDesc->Length);
                  TempBlockDesc->Union.DataBlock = (EFI_PHYSICAL_ADDRESS) (UINTN) RelocPtr;
               }

      // Next descriptor
      TempBlockDesc++;
    }
     ...
     CopyMem ((VOID *) DestPtr, (VOID *) (UINTN) (CurrentBlockDesc->Union.DataBlock), (UINTN)CurrentBlockDesc->Length);
     DestPtr += CurrentBlockDesc->Length;
```

**Listing 7:** CapsuleDataCoalesce code continued

### 4.1.1 Coalescing Exploitation Difficulties

The most obvious exploitation approach is to supply a Capsule with CapsuleSize large enough to force CapsuleSize + DescriptorSize to overflow (Bug 1). Then the process of coalescing the huge capsule will overflow the intended coalescing area and corrupt the address space. Figure 4 demonstrates this approach.

However, this most obvious approach was insufficient on the MinnowBoard. When the overflow began writing into the address space gap described in Section 4.1, the writes would silently fail. Although the destination pointer for the memory copy operation continued to proceed upwards despite these invalid writes, a timeout associated with the failed write slowed the process down to a prohibitively slow pace. A different approach was considered.

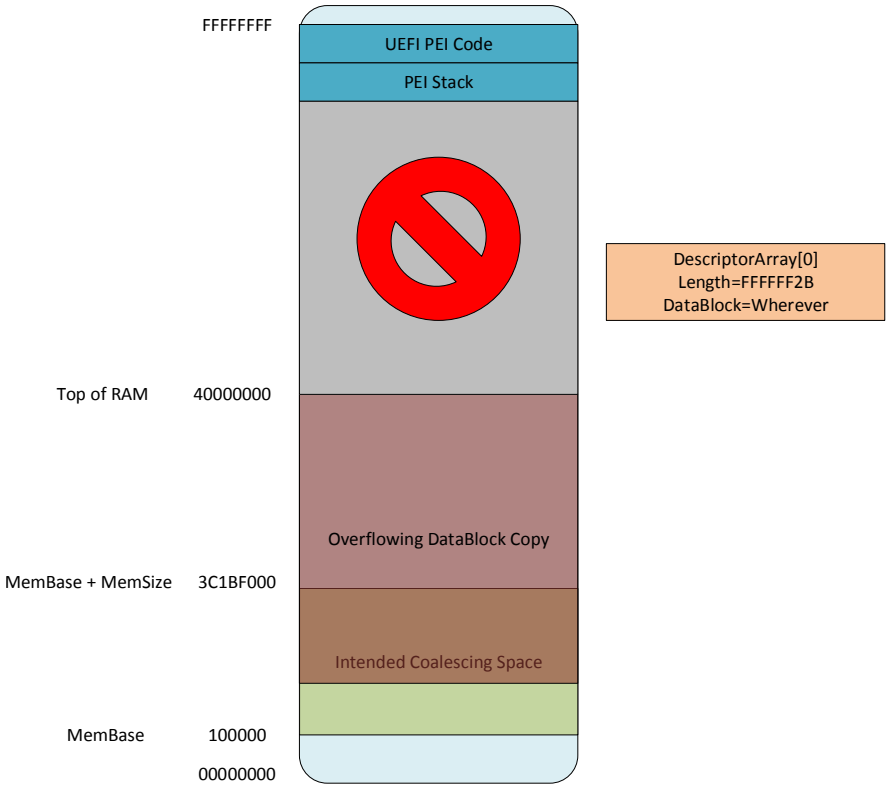


**Figure 4:** First Attempt at Coelescing Exploitation fails due to the address space gap

### 4.1.2 Descriptor Overwrite Approach

It was necessary to devise a way to overwrite a function pointer in the high portion of the address space without touching the address space gap. We chose a multistage approach that would first corrupt the descriptor array, so that DestPtr would be adjusted by a corrupted descriptor length value. This approach allows exact control of DestPtr on a subsequent block copy.

One major hurdle stood in the way of the descriptor overwrite approach; before the coalescing block copy operations begins, the descriptor array is relocated to the bottom of the address space. This ensures that the descriptor array is always out of the way of any block copy operations. Additional tricks were needed to proceed further with this approach.

### 4.1.3 Optimization Tricks

An examination of the CopyMem implementation yielded a clever trick that could be abused for the descriptor overwrite approach.

```
CopyMem (
  OUT VOID       *DestinationBuffer,
  IN CONST VOID  *SourceBuffer,
  IN UINTN       Length
  )
{
  ...
  if (DestinationBuffer == SourceBuffer) {
    return DestinationBuffer;
  }
  return InternalMemCopyMem (DestinationBuffer, SourceBuffer, Length);
}
```

**Listing 8:** CopyMem implementation

Note in Listing 8 that CopyMem is optimized so that if DestinationBuffer and SourceBuffer are equal, the function will automatically exit successfully. Hence a huge CopyMem can be performed that will skip over the address space gap, and DestPtr will subsequently be increased by a huge value. Using this approach allowed DestPtr to be wrapped to the bottom of the address space where the relocated descriptor array had been placed.

Also note that this CopyMem optimization abuse can not be used to set DestPtr directly at any function pointers high in the address space. This requirement results from the IsOverlapped check described in code Listing 7. The IsOverlapped check validates that the current block copy operation will not clobber the current block or any other remaining data blocks. Because we have explicitly set DestPtr equal to CurrentBlockDesc's Data member in order to abuse the CopyMem optimization, the blocks necessarily overlap. However, if we examine the IsOverlapped implementation, we see a way out.

```
BOOLEAN
IsOverlapped (
  UINT8     *Buff1,
  UINTN     Size1,
  UINT8     *Buff2,
  UINTN     Size2
  )
{
  //
  // If buff1's end is less than the start of buff2, then it's ok.
  // Also, if buff1's start is beyond buff2's end, then it's ok.
  //
  if (((Buff1 + Size1) <= Buff2) || (Buff1 >= (Buff2 + Size2))) { <= Bug 4
    return FALSE;
  }

  return TRUE;
}
```

**Listing 9:** IsOverlapped implementation

IsOverlapped will erroneously return false if an integer overflow is induced by Buff1 + Size1 (Bug 4). Thus the CopyMem optimization trick cannot be used to set DestPtr directly at the high portion of the address space, as this would not induce an integer overflow in the IsOverlapped calculation and IsOverlapped would return true. However, this works perfectly to wrap DestPtr to the bottom of the address space, which then provides the possibility to overwrite the relocated descriptor array.

### 4.1.4 Coalesce Exploitation Success

The return address for the CopyMem function proved to be a viable target to get control of the instruction pointer. This was accomplished using the following steps which ultimately allowed for a surgical write-what-where exploitation primitive. The steps are also depicted visually in Figures 5 through 8.

1. DescriptorArray[0] contains a superficial Capsule Header needed for sanity checking purposes and is copied normally. See Figure 5.

2. DescriptorArray[1] abuses the CopyMem optimization trick and IsOverlapped integer overflow to wrap DestPtr around the address space. See Figure 6.

3. DescriptorArray[2] overwrites its own Length value, so that DestPtr can be arbitrarily adjusted. See Figure 7.

4. DescriptorArray[3] overwrites the return address for the CopyMem function. Control is gained here. See Figure 8.

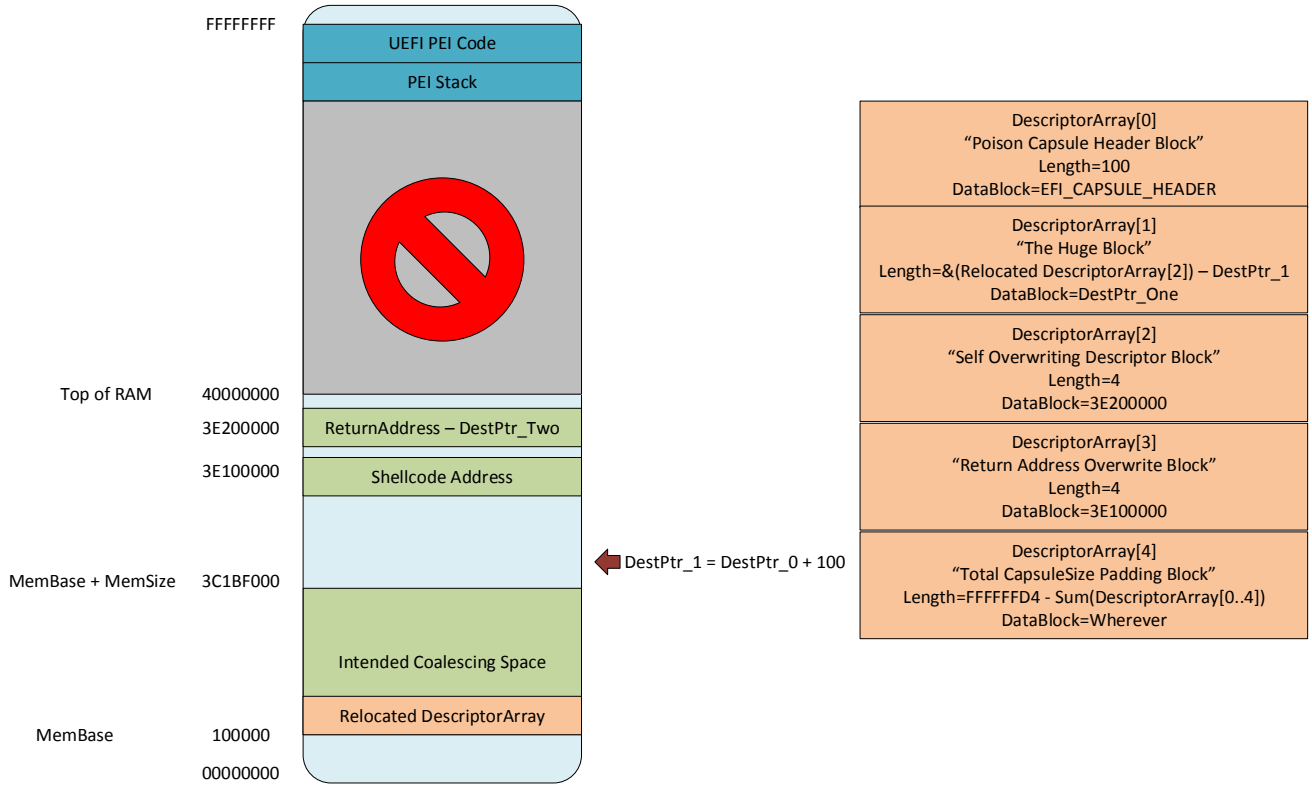


**Figure 5:** DescriptorArray[0] contains a superficial Capsule Header needed for sanity checking purposes and is copied normally.

## 4.2 Envelope Exploitation

Exploiting the vulnerability in the parsing of envelope of the capsule proved to be challenging as well. Consider the code in Listing 10 that writes to the underallocated LbaCache buffer (Bug 3).

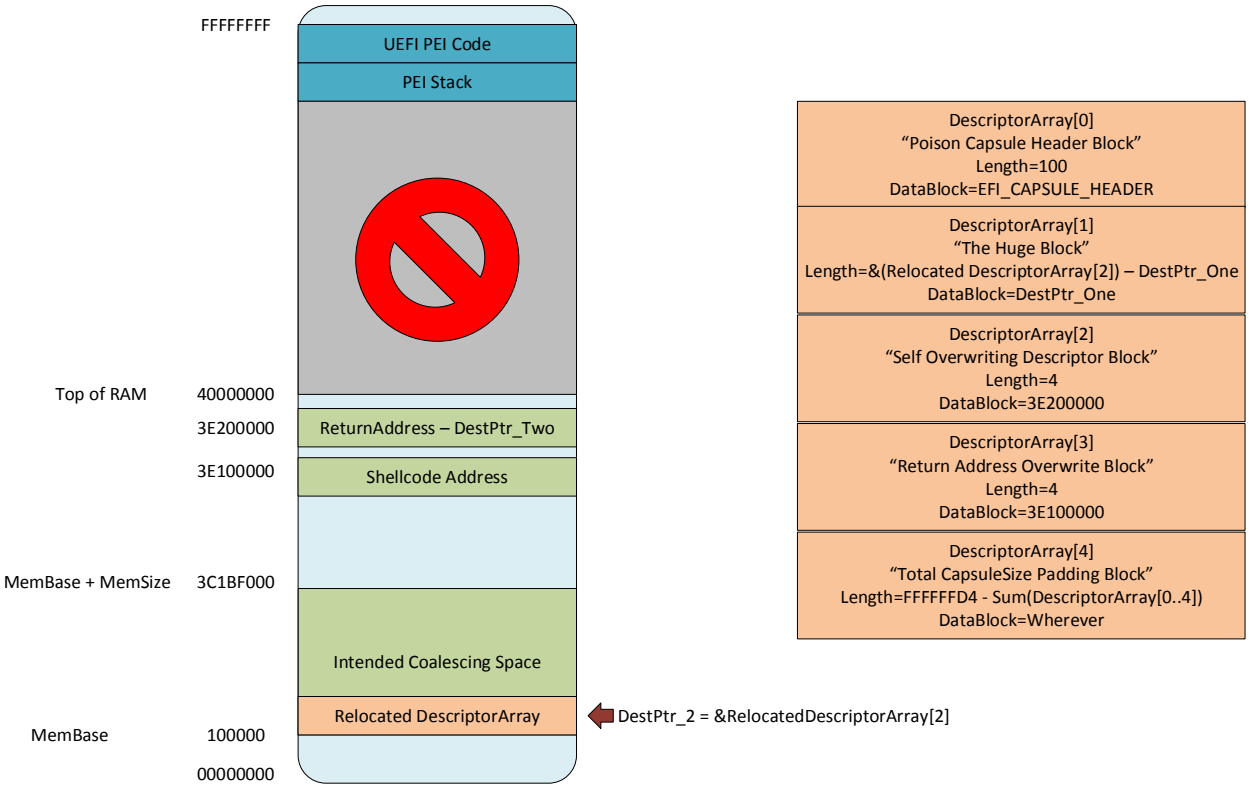


**Figure 6:** DescriptorArray[1] abuses the CopyMem optimization trick and IsOverlapped integer overflow to wrap DestPtr around the address space.

```
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++;
  }
}
```

**Listing 10:** ProduceFVBProtocolOnBuffer code continued.

Recall that NumBlocks had to be set very large in order to induce the overflow in the LbaCache allocation (Bug 3). Unfortunately this also means that the above loop will end up corrupting the majority of the address space and destabilize the system if allowed to run to completion. Another complication is the discovery that LbaCache was being allocated below the FvbDev structure. This meant that the overwriting loop would end up corrupting the LbaCache pointer, further complicating the progression of the corruption. This issue is illustrated in Figure 9. Lastly, note that the corruption occurs via a series of pairs of 4 byte writes. One of the writes, PtrBlockMapEntry's Length member, is attacker controlled. However, the other is the write

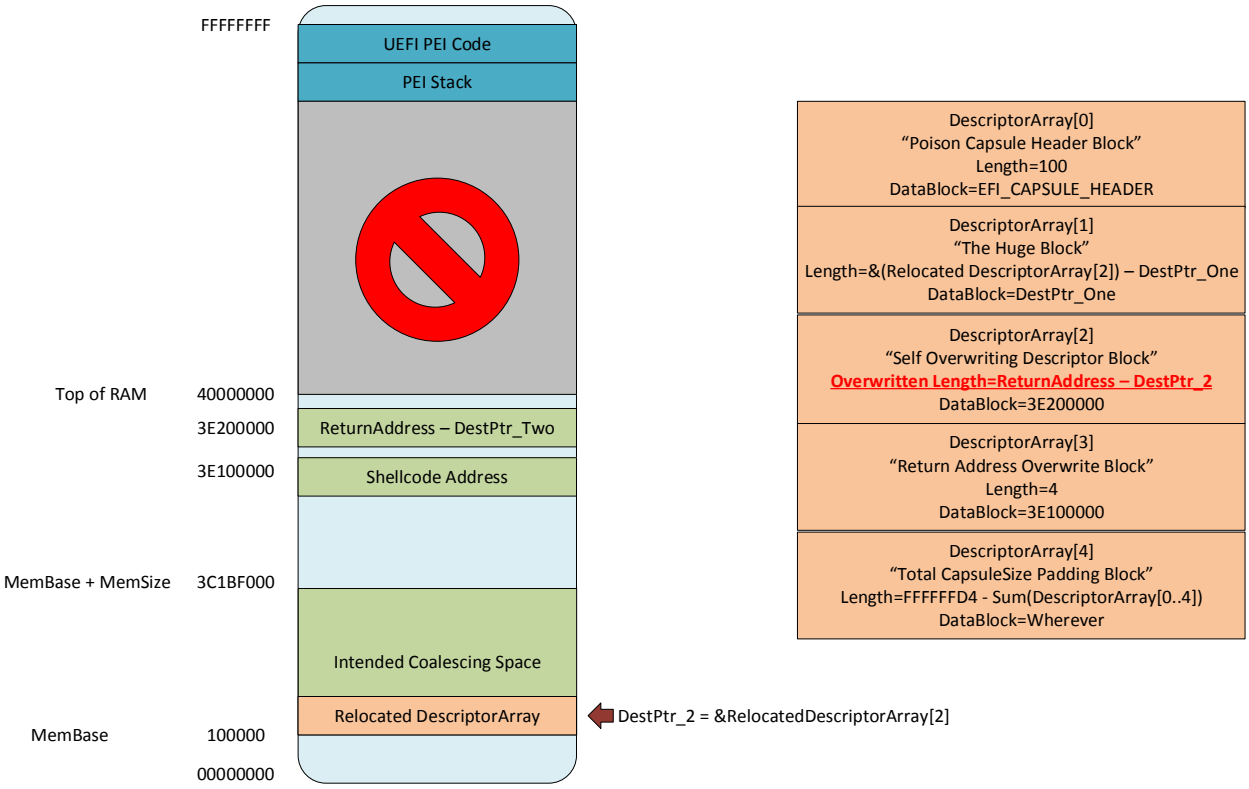


**Figure 7:** DescriptorArray[2] overwrites its own Length value, so that DestPtr can be arbitrarily adjusted.

of LinearOffset. LinearOffset is incremented during every iteration of the loop and thus is only partially attacker controlled.

The most pressing constraint is the non terminating nature of the corrupting loop. In order to escape the loop, it was necessary to overwrite the loop code itself. However, as the values being written are not completely attacker controlled, it was a matter of brute force to determine what values of PtrBlockMapEntry's Length member would lead to overwriting the loop code with coherent x86 instructions. Figure 10 shows an attempt that overwrites the top of the loop's basic block with x86 instructions that are not advantageous to an attacker. Figure 11 shows a Length value discovered by our brute force script that leads to attacker advantageous instructions overwriting the loop code.

## 4.3 Exploitation From Windows 8

Two conditions are necessary for the exploitation of the vulnerabilities.

- The ability to instantiate the capsule update process.
- The ability to stage arbitrary data at certain physical addresses.

If the attacker already has ring 0 code execution, these conditions are trivially met. The attacker can call the capsule update Runtime Service directly to meet the first condition. The second condition can be met via a number of kernel APIs that allow access to physical memory, such as MmAllocateContiguousMemory[5], or via direct page table manipulation.

The attack is also possible from a privileged user in ring 3. The introduction of the userland EFI variable API in Windows 8 inadvertently exposes the capsule update process to userland. This follows from the fact

[5] `http://msdn.microsoft.com/en-us/library/windows/hardware/ff554460(v=vs.85).aspx`

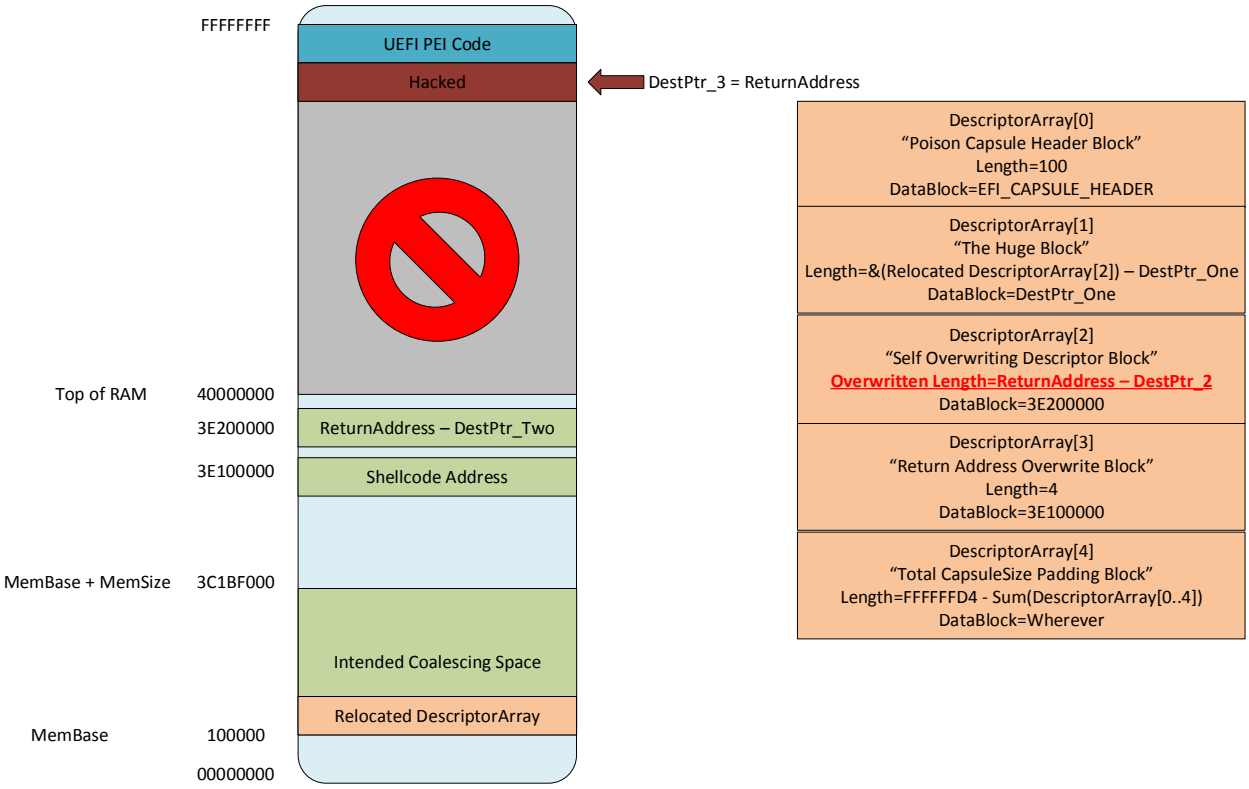


**Figure 8:** DescriptorArray[3] overwrites the return address for the CopyMem function. Control is gained here.

that capsule update is automatically initiated by the firmware if the "CapsuleUpdateData" EFI variable exists during a warm reset of the system.

A privileged userland process also has several ways to surmount the second attack requirement. We do not assume the attacker needs their own kernel driver signing key. However, given our attack model assumes a privileged user, such a user is able to install any authenticode signed kernel drivers onto the system. There exist such drivers that will arbitrarily modify the content of physical memory on a users behalf[6]. The technique of using known-vulnerable, but signed, 3rd party drivers to perform exploits or actions on the attacker's behalf, has been discussed since Windows Vista's inception[19] and has been used in the wild by malware[17].

A more direct approach to satisfy the second requirement is to further abuse the Windows 8 EFI variable interface. We found that the creation of many large sized EFI variables eventually resulted in attacker controlled data residing at predictable physical addresses. This resulted from the EFI variables themselves being stored on the SPI flash chip, and the contents of the SPI flash chip being mirrored to the address space during the bootup of the system. Using this *variable spray* technique we were able to stage the necessary payloads at predictable physical addresses and reliably exploit the coalescing vulnerability using only the Windows 8 userland EFI variable API.

Exploitation of the capsule envelope vulnerability may not be possible from Windows 8 userland. The majority of consumer platforms we analyzed executed their DXE phase in 64 bit mode. Hence as described in section 4.2, it is necessary for the attacker to control a contiguous 2 GB part of the address space in order to induce the underlying integer overflow. We are not aware of any methods by which a userland attacker could reliably stage a physically contiguous 2GB region. Another complicating factor is the only semi-controlled nature of the corruption. The attacker is dependent on finding some values that will allow overwriting the non-terminating loop with attacker advantageous instructions. Typically these instructions

[6] `http://rweverything.com/`

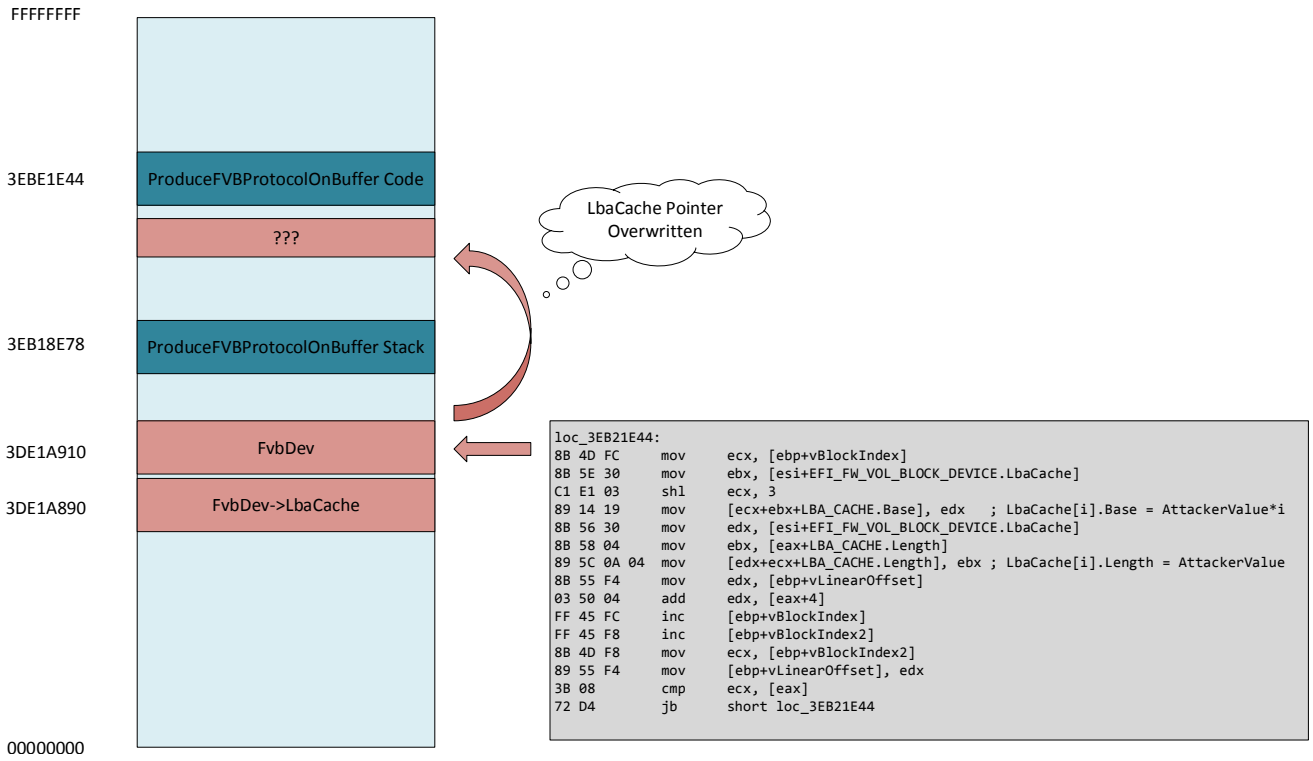


**Figure 9:** The LbaCache pointer is corrupted, further complicating the overwrite.

will take the form of jumping or calling to a non-corrupted address. Exploitation of the envelope vulnerability from userland also dictates that a specific target physical address will need to be attainable from a Windows 8 userland process. This may not be possible depending on the target address, for instance if that physical address is typically allocated to the kernel.

# 5 Leveraging The Attack

Successful exploitation of these vulnerabilities allows the attacker to gain code execution in the early boot environment. In this environment, the SPI flash and SMRAM are still necessarily unlocked. Thus the attacker is able to make SPI write cycles to the SPI flash, allowing him to permanently persist in the platform firmware. Because the platform firmware is responsible for instantiating SMM, the attacker is able to use this approach to arbitrarily inject code into SMM as well. Note that this means the attackers injections would survive even an operating system reinstallation! In fact, as shown in [3] it would then be possible for the attacker to also persist across firmware update attempts.

Another interesting possibility is for the attacker to avoid reflashing the firmware directly, and instead continually exploit the firmware during reset of the system. As described in section 4.3, an attacker can store his payload in a persistent EFI non-volatile variable. During a warm reset of the system, the UEFI code will then be exploited when attempting to process the lying in wait evil capsule. Once the attacker has control, he can make arbitrary insertions into SMM, and then let the system boot up normally. In this way the attacker can establish a presence in SMM during each warm reset of the system, without having to make direct SPI flash writes to the UEFI code. This approach may be desirable if the attacker wishes to further obfuscate his presence.

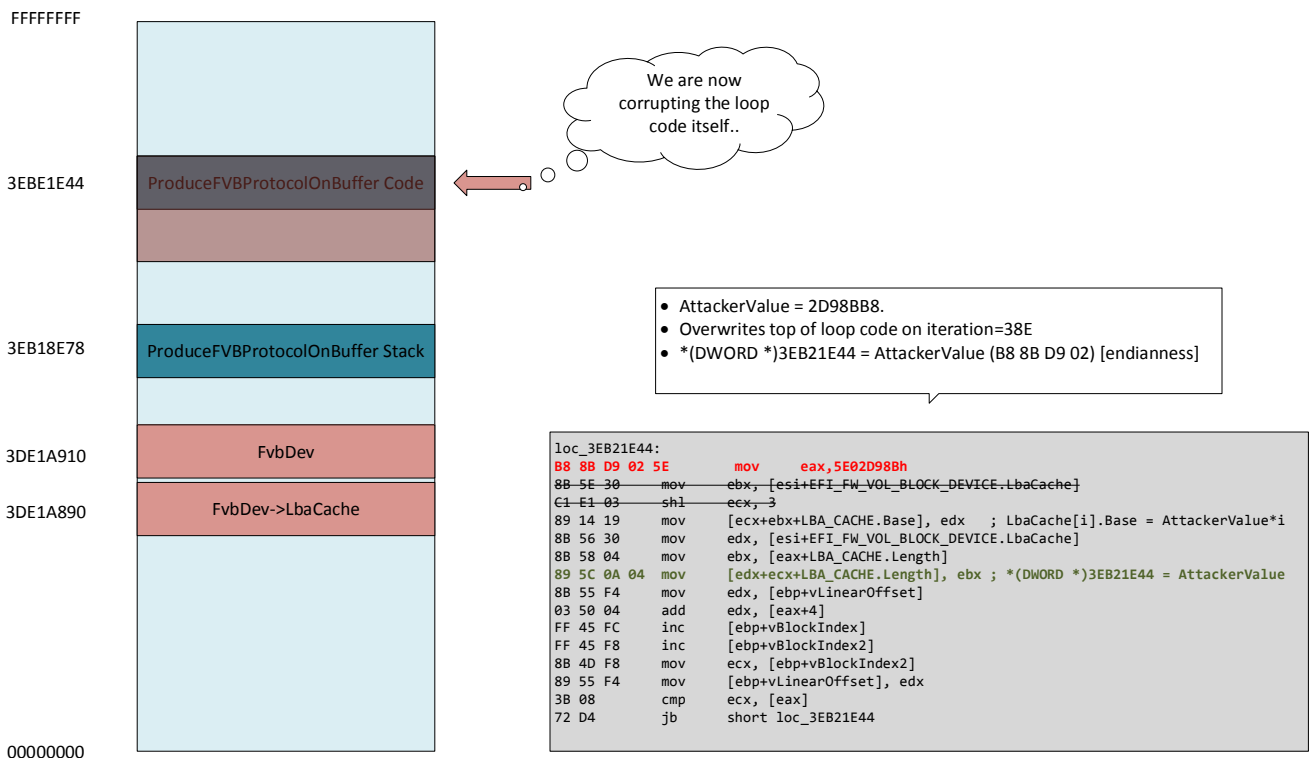


**Figure 10:** Loop code overwritten with useless instructions.

# 6 User Experience

Exploiting the capsule update vulnerabilities requires rebooting the system. An attacker wishing to remain stealthy could schedule the attack to occur when the system is naturally rebooting for patches. After the reboot has occurred, and the attacker has pivoted control to a firmware injecting shellcode, another reboot of the system should immediately take place. Because the vulnerabilities take place before graphics have been initialized, the victim may only notice a longer than usual reboot time.

# 7 Affected Systems

Determining which vendors were affected was a non-trivial problem. Theoretically the UEFI open source reference implementation should be widely utilized by both OEMs and Independent BIOS Vendors (IBVs). Thus the capsule update vulnerabilities should be widespread. In practice there is large variance to the degree that OEMs/IBVs utilize the reference implementation. This problem is further described by [16], which points out that firmware implementations vary widely even within the same OEM. Due to this, it is necessary to consider the exploitability of systems on a case by case basis. Before we can begin with a case study, analysis techniques that help determine exploitability are introduced.

## 7.1 OEM Firmware Instrumentation

As mentioned in Section 4, the lack of debugging capability for firmware level code is a significant hurdle. Without debugging capability, the vulnerability of a particular system must be determined using only static analysis. Given the complexity of UEFI and its tendency for indirect calling, this approach proved difficult. To work around this limitation, we used QEMU to instrument the OEM firmware using the following steps.

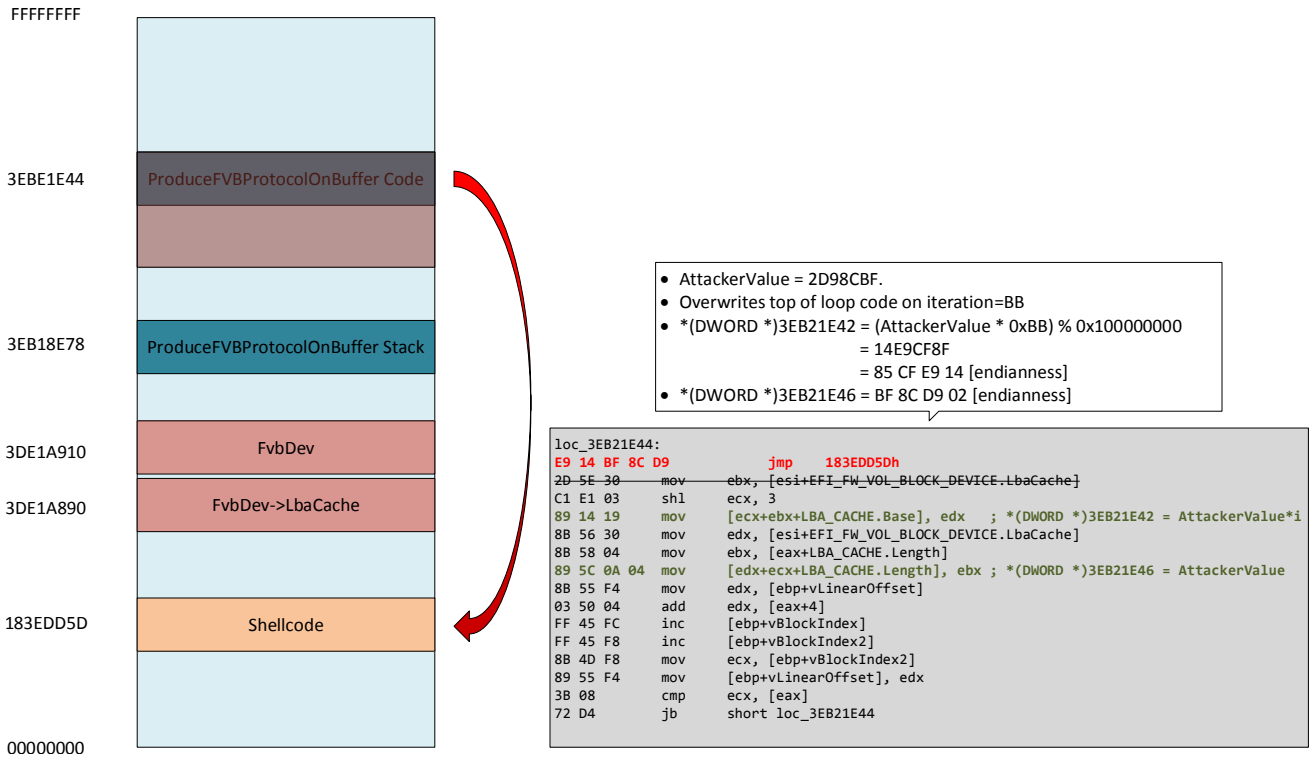


**Figure 11:** Loop code overwritten with jump to shellcode.

- The OEM Firmware was dissected into its individual EFI executables using EFIPWN[7].
- The EFI executables responsible for Capsule Coalescing (PEI Phase) and Capsule Processing (DXE Phase) were identified using Guid matching and bindiff[8].
  - Capsule coalescing code was usually located in CapsulePEI.
  - Capsule processing code was usually located in DXECORE.
- The relevant binary modules were then inserted into the UEFI Open Virtual Machine Firmware[1][9]
- Because OVMF does not have built in support for capsule update, the PlatformPeim module provided by OVMF was modified to call the capsule update interface exported by the OEM's binary module.
- QEMU was then used to boot the modified OVMF firmware.
- Debugging of the OEM capsule binary modules was now possible via QEMU's gdb stub.

## 7.2 HP EliteBook 2540p F23 Case Study

As an example of determining the vulnerability of a specific OEM system, we consider the capsule coalescing routine of the HP EliteBook 2540p at BIOS (UEFI) revision F23. We discovered the capsule coalescing code to be very similar to the code described in Listing 4, with the following relevant differences.

- CapsuleSize + 8 is compared to MemorySize, as opposed to CapsuleSize + DescriptorSize. Hence Bug 1 is present in a different form.

[7] `https://github.com/G33KatWork/EFIPWN`
[8] `http://www.zynamics.com/bindiff.html`
[9] OVMF is an open source virtual machine firmware included in the UEFI open source reference implementation.

- An additional sanity check is made that for each entry in the descriptor array, descriptor.DataBlock + descriptor.Length does not integer overflow[10].
- EDK1 style descriptors are used, which include a 4 byte signature and a checksum.[11] The DataBlock and Length fields are identical.

To demonstrate the vulnerability of the 2540p, we built a descriptor list that explicitly instantiated Bug 2[12] and Bug 4[13]. Both of these bugs were present in the 2540p. Consider the *matryoshka* style descriptor array configuration outlined by Figure 12. In this configuration, the sum of the descriptor length values will overflow the 32 bit CapsuleSize integer, and hence pass the sanity comparison with MemorySize. Also note that although it is sanity checked that Descriptor[0].Length + Descriptor[0].DataBlock does not overflow, it is still possible that DestPtr + Descriptor[0].Length will overflow. This is in fact the case since we explicitly set Descriptor[0] to be low in memory and have a huge length size, and DestPtr is always high in the address space due to the coalescing function design. Hence we can abuse Bug 4 in the IsOverlapped check to proceed with the block copy operation of Descriptor[0]. This copy will corrupt the address space and allow the attacker to hijack control of the update operation.[14]

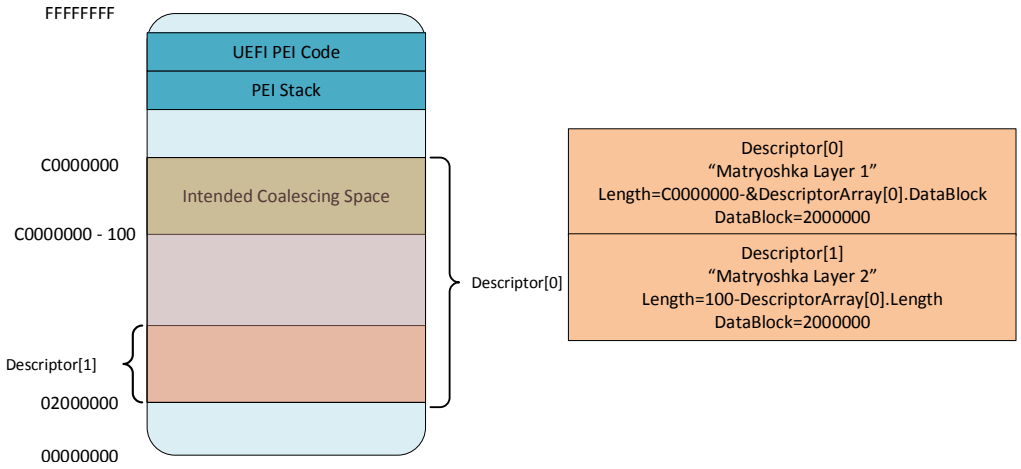


**Figure 12:** The Matryoshka descriptor configuration. Sum of length values overflows 32 bit Integer.

## 7.3 General Observations Regarding Affected Systems

We believe the vulnerabilities presented in this paper to be widespread among UEFI systems. However, it is difficult to determine the exact pervasiveness of the vulnerabilities due to the need to evaluate firmware implementations on a case by case basis. Some variant of the capsule coalescing vulnerability was present in the majority of systems we evaluated with static analysis. Interestingly, although the OEM coalescing code often demonstrated variations from the reference implementation code, the ProduceFVBProtocolOnBuffer code appeared to be identically copied in all of the systems we looked at. Thus the code associated with bug 3[15] was present on all of the UEFI implementations we analyzed. However, as we did not have debug access, or even physical access[16] to many systems we evaluated, it is impossible to produce a complete list of affected systems without self-reporting by the vendors. For instance, although the vulnerable code may be present, it may be vestigial and not actually utilized during the update process.

[10] An important sanity check that the reference implementation lacked
[11] Neither of these 2 additional fields matters in practice
[12] Integer overflow in capsule length summation
[13] Integer overflow in IsOverlapped check
[14] In the case of the Elitebook 2540p, 4GB RAM is standard so there is no dead space in the address space as had to be overcome on the MinnowBoard.
[15] Integer overflow in LbaCache allocation
[16] We just downloaded the firmware images from the OEM websites.

# 8 Vendor Response

Intel and CERT were notified of the envelope parsing vulnerability on November 22nd 2013, and of the coalescing vulnerabilities on December 4th 2014. Intel then reached out to IBVs and OEMs to attempt to discern if they were affected. Information about which vendors are affected and what systems should be patched is conveyed in CERT VU #552286. The disclosure of these vulnerabilities ultimately led to the formation of a UEFI Security Response Team. The vulnerabilities were patched in the UDK2014 reference implementation release[10].

# 9 Recommendations

The authors have several recommendations regarding locking down the UEFI capsule update interface and the Runtime Interface as a whole.

- The UEFI Reference Implementation should be more thoroughly audited. The existence of easy to find integer overflows in security critical code is disturbing.
- Capsule coalescing seems unnecessary on modern system's with plentiful RAM as firmware capsules are usually under 16 MB. Instead the firmware capsule could assumed to be already contiguous in memory. This would eliminate much of the complicated and buggy code from the firmware update process. If memory constraints are a valid concern [17], the firmware update process could be instantiated from a boot loader.
- The decision to expose the Variable portion of the Runtime Services to userland in Windows 8 should be more closely evaluated from a security standpoint. On the one hand, this decision minimizes the amount of code that needs to be loaded into the kernel, as now userland processes can perform important system configuration. On the other hand, userland access to these variables opens up additional attack surface that is accessible from ring 3.
- An option for a physical presence test should be added to the firmware update process[18]. Although many organizations will opt out of this option so that they can remotely update firmware without user interaction, organizations with a greater security focus could opt in to this requirement.

# 10 Related Work

Invisible Things Lab presented the first memory corruption attack against a BIOS update[20]. In their attack, an integer overflow in the rendering of a customizable bootup splash screen was exploited to gain control over the boot up process before the BIOS locks were set. This allowed the BIOS to be reflashed with arbitrary contents. The authors of this paper have also presented an attack against the coalescing operation of some Dell legacy BIOSes[15].

# 11 Conclusion

In this paper we have demonstrated that although UEFI provides additional security features, it has also provided additional attack surface that may be exploited. Furthermore, some of this attack surface is exposed to Windows 8 userland processes by the Windows 8 firmware environment variable API. Despite increased focus on protecting the integrity of the platform, vulnerabilities introduced by UEFI may allow an attacker to compromise the platform firmware and attain permanent control of SMM. Although the authors believe that UEFI is ultimately a step in the right direction towards securing the platform, more work needs to be done on evaluating the security of the features it provides.

[17] Such as on an embedded system

[18] Toggled perhaps through the BIOS configuration screen

## 12 Acknowledgments

Thanks to the Intel Product Security Incident Response Team (PSIRT) for helping coordinate these vulnerabilities with affected OEMs and IBVs.

## References

[1] How to build OVMF. http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=How_to_build_OVMF. Accessed: 06/13/2014.

[2] MinnowBoard. http://www.minnowboard.org. Accessed: 06/13/2014.

[3] J. Butterworth, C. Kallenberg, and X. Kovah. Bios chronomancy: Fixing the core root of trust for measurement. In *BlackHat*, Las Vegas, USA, 2013.

[4] CERT. VU #758382. http://www.kb.cert.org/vuls/id/758382. Accessed: 06/13/2014.

[5] Intel Corporation. Intel I/O Controller Hub 9 (ICH9) Family Datasheet. http://www.intel.com/content/www/us/en/io/io-controller-hub-9-datasheet.html. Accessed: 10/01/2013.

[6] Intel Corporation. Intel Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification. http://www.intel.com/content/dam/doc/reference-guide/efi-dxe-cis-v091.pdf. Accessed: 06/13/2014.

[7] Intel Corporation. Intel Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification. http://www.intel.com/content/dam/doc/reference-guide/efi-pei-cis-v091.pdf. Accessed: 06/13/2014.

[8] Intel Corporation. TianoCore. http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=Welcome. Accessed: 06/13/2014.

[9] Intel Corporation. UEFI Development Kit 2010. http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=UDK2010. Accessed: 06/13/2014.

[10] Intel Corporation. UEFI Development Kit 2010. http://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/. Accessed: 06/13/2014.

[11] Microsoft Corporation. MSDN SetFirmwareEnvironmentVariable. http://msdn.microsoft.com/en-us/library/windows/hardware/ms724934(v=vs.85).aspx. Accessed: 06/13/2014.

[12] j00ru. Defeating Windows Driver Signature Enforcement #1: default drivers. http://j00ru.vexillium.org/?p=1169. Accessed: 06/13/2014.

[13] j00ru. Defeating Windows Driver Signature Enforcement #2: CSRSS and thread desktops. http://j00ru.vexillium.org/?p=1393. Accessed: 06/13/2014.

[14] j00ru. Defeating Windows Driver Signature Enforcement #3: The Ultimate Encounter. http://j00ru.vexillium.org/?p=1455. Accessed: 06/13/2014.

[15] C. Kallenberg, J. Butterworth, X. Kovah, and C. Cornwell. Defeating Signed BIOS Enforcement. In *EkoParty*, Buenos Aires, 2013.

[16] C. Kallenberg, C. Cornwell, X. Kovah, and J. Butterworth. Setup For Failure: More Ways to Defeat SecureBoot. In *Hack In The Box Amsterdam*, Amsterdam, 2014.

[17] MN. Analysis of Uroburos, using WinDbg. https://blog.gdatasoftware.com/blog/article/analysis-of-uroburos-using-windbg.html. Accessed: 06/16/2014.

[18] Phoenix. Efi Runtime Services. http://wiki.phoenix.com/wiki/index.php/EFI_RUNTIME_SERVICES. Accessed: 06/13/2014.

[19] J. Rutkowska and A. Tereshkin. IsGameOver() Anyone? In *BlackHat*, Las Vegas, USA, 2007.

[20] R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. In *BlackHat*, Las Vegas, USA, 2009.