Abusing Exceptions for Code Execution, Part 1

billdemirkapi.me/exception-oriented-programming-abusing-exceptions-for-code-execution-part-1

Bill Demirkapi February 14, 2022

A common offensive technique used by operators and malware developers alike has been to execute malicious code at runtime to avoid static detection. Often, methods of achieving runtime execution have focused on placing arbitrary code into executable memory that can then be executed.

In this article, we will explore a new approach to executing runtime code that does not rely on finding executable regions of memory, but instead relies on abusing existing trusted memory to execute arbitrary code.

Background

Some common methods of runtime execution include:

1. Allocating executable memory at runtime.
2. Abusing Windows Section objects.
3. Abusing existing RWX (read/write/execute) regions of memory allocated by legitimate code.
4. Loading legitimate binaries that include an RWX PE section that can be overwritten.

One common pattern in all these methods is that they have always had a heavy focus on placing arbitrary shellcode in executable regions of memory. Another technique that has not seen significant adoption in the malware community due to its technical complexity is Return-Oriented Programming (ROP).

As a brief summary, ROP is a common technique seen in memory corruption exploits where an attacker searches for “snippets of code” (gadgets) inside binaries that perform a desired instruction and soon after return to the caller. These “gadgets” can then be used in a chain to perform small operations that add up to a larger goal.

Although typically used for memory corruption exploits, there has been limited use in environments where attackers already have code execution. For example, in the Video Game Hacking industry, there have been open-source projects that allow cheaters to abuse ROP gadgets to execute simplified shellcode for the purposes of gaining an unfair advantage in a video game.

The greatest limitation with ROP for runtime execution however is that the instruction set of potential gadgets can be extremely limited and thus is challenging to port complex shellcode to.

As an attacker that already has execution in an environment, ROP is an inefficient method of performing arbitrary operations at runtime. For example, assume you are given the following assembly located in a legitimate binary:

```
imul eax, edx
add eax, edx
ret
```

Using ROP, an attacker could only abuse the `add eax, edx` instruction because any previous instructions are not followed by a return instruction.

On a broader scale, although legitimate binaries are filled with a variety of different instructions performing all-kinds of operations, the limitations of ROP prevent an attacker from using most of these legitimate instructions.

Continuing the example assembly provided, the reason an attacker could not abuse the initial `imul eax, edx` instruction without corrupting their output is because as soon as the `imul` instruction is executed, the execution flow of the program would simply continue to the next `add eax, edx` instruction.

Theory: Runtime Obfuscation

I propose a new method of abusing legitimate instructions already present in trusted code, **Exception Oriented Programming**. Exception Oriented Programming is the theory of chaining together instructions present in legitimate code and “stepping over” these instructions one-by-one using a single step exception to simulate the execution of arbitrary shellcode.

As a general method, the steps to performing Exception Oriented Programming given arbitrary shellcode are:

1. Setup the environment such that the program can intercept single step exceptions.
2. Split each assembly instruction in the arbitrary shellcode into their respective assembled bytes.
3. For each instruction, find an instance of the assembled bytes present in any legitimate module and store the memory location of these legitimate instructions.
4. Execute any code that will cause an exception that the exception handler created in Step 1 can intercept. This can be as simple as performing a call instruction on an `int3` (0xCC) instruction.
5. In the exception handler, set the single step flag and set the instruction pointer register to the location of the next legitimate instruction for that shellcode.
6. Repeat Step 5 until all instructions of the shellcode are executed.

The largest benefit to this method is that Exception Oriented Programming has significantly less requirements than Return Oriented Programming for an attacker that already has execution capabilities.

Next, we will cover some practical implementations of this theory. Although the implementations you will see are operating system specific, the theory itself is not restricted to one operating system. Additionally, implementation suggestions will stick strictly to documented methods, however it may be possible to implement the theory via undocumented methods such as directly hooking `ntdll!KiUserExceptionDispatcher`.

Vectored Exception Handlers

In this section, we will explore how to use Vectored Exception Handlers (VEH) to implement the EOP theory. Vectored Exception Handlers are an extension to Structured Exception Handling on Windows and are not frame-based. VEH will be called for unhandled exceptions regardless of the exception's location.

The flow for the preparation stage of this implementation is as follows:

1. The application will register a VEH via the `AddVectoredExceptionHandler` Windows API.
2. The application will split each instruction of the given shellcode using any disassembler. For the example proof-of-concept, we will use the Zydis disassembler library.
3. For each split instruction, the application will attempt to find an instance of that instruction present in the executable memory of any loaded modules. These memory locations will be stored for later use by the exception handler.
4. The application will finish preparing by finding any instance of an `int3` instruction (a single 0xCC byte). This instruction will be stored for use by the exception handler and is returned to the caller which will invoke the arbitrary shellcode.

Once the necessary memory locations have been found, the caller can invoke the arbitrary shellcode by executing the `int3` instruction that was returned to them.

1. Once the caller has invoked the int3 instruction, the exception handler will be called with the code `STATUS_BREAKPOINT`. The exception handler should determine if this exception is for executing arbitrary shellcode by comparing the exception address with the previously stored location of the `int3` instruction.

2. If the breakpoint is indeed for the arbitrary shellcode, then the exception handler should:
    1. Retrieve the list of legitimate instructions needed to simulate the arbitrary shellcode.
    2. Store these instructions in thread-local storage.
    3. Set the instruction pointer to the first legitimate instruction to execute.
    4. Set the Trap flag on the FLAGS register.
    5. Continue execution.
3. The rest of the instructions will cause a `STATUS_SINGLE_STEP` exception. In these cases, the exception handler should:
    1. Retrieve the list of legitimate instructions to execute from the thread-local storage.
    2. Set the instruction pointer to the next legitimate instruction's memory location.
    3. If this instruction is *not* the last instruction to execute, set the Trap flag on the FLAGS register. Otherwise, do *not* set the Trap flag.
    4. Continue execution.

Assuming the shellcode ends with a return instruction, eventually the execution flow will be gracefully returned to the caller. A source code sample of Exception Oriented Programming through Vectored Exception Handlers is provided in a later section.

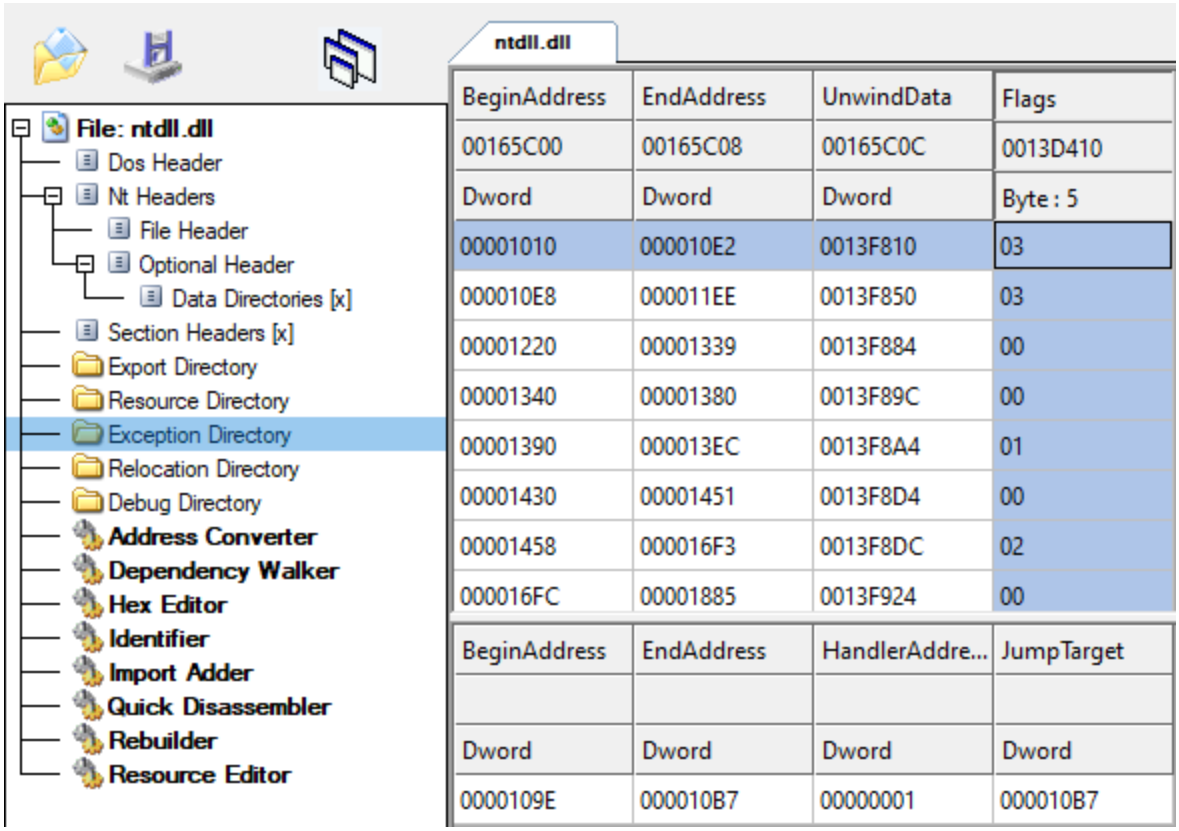
Structured Exception Handlers

Although Vectored Exception Handlers are great, they're not exactly stealthy. For example, an anti-virus could use user-mode hooks to detect when vectored exception handlers are registered. Obviously there are plenty of ways to bypass such mitigations, but if there are stealthier alternatives, why not give them a try?

One potential path I wanted to investigate for Exception Oriented Programming was using generic Structured Exception Handling (SEH). Given that VEH itself is an extension to SEH, why wouldn't frame-based SEH work too? Before we can dive into implementing Exception Oriented Programming with SEH, it's important to understand how SEH works.

```
void my_bad_code() {
    __try {
        __int3;
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Exception handler called!");
    }
}
```

Let's say you surround some code with a try/except SEH block. When an exception occurs in that code, how does the application know what exception handler to invoke?



Exception Directory of ntdll.dll

Nowadays SEH exception handling information is compiled into the binary, specifically the exception directory, detailing what regions of code are protected by an exception handler. When an exception occurs, this table is enumerated during an "unwinding process", which checks if the code that caused the exception or any of the callers on the stack have an SEH exception handler.

An important principle of Exception Oriented Programming is that your exception handler must be able to catch exceptions in the legitimate code that is being abused. The problem with SEH? If a function is already protected by an SEH exception handler, then when an exception occurs, the exception may never reach the exception handler of the caller.

This presents a challenge for Exception Oriented Programming, how do you determine whether a given function is protected by an incompatible exception handler?

Fortunately, the mere presence of an exception handler does not mean a region of code cannot be used. Unless the function for some reason would create a single step exception during normal operation or the function has a "catch all" handler, we can still use code from many functions protected by an exception handler.

To determine if a region of memory is compatible with Exception Oriented Programing:

1. Determine if the region of memory is registered as protected in the module's exception directory. This can be achieved by directly parsing the module or by using the function RtlLookupFunctionEntry, which searches for the exception directory entry for a given address.
2. If the region of memory is not protected by an exception handler (aka RtlLookupFunctionEntry returns NULL), then you can use this region of memory with no problem.
3. If the region of memory is protected by an exception handler, you must verify that the exception handler will not corrupt the stack. During the unwinding process, functions with an exception handler can define "unwind operations" to help clean up the stack from changes in the function's prolog. This can in turn corrupt the call stack when an exception is being handled.
    1. To avoid this problem, check if the unwind operations contains either the UWOP_ALLOC_LARGE operation or the UWOP_ALLOC_SMALL operation. These were found to cause direct corruption to the call stack during testing.

Once compatible instruction locations are found within legitimate modules, how do you actually perform the Exception Oriented Programming attack with SEH? It's surprisingly simple.

With SEH exception handling using a try except block, you can define both an exception filter and the handler itself. When an exception occurs in the protected try except block, the exception filter you define determines whether or not the exception should be passed to the handler itself. The filter is defined as a parameter to the __except block:

```
void my_bad_code() {
    __try {
        __int3;
    } __except(MyExceptionFilter()) {
        printf("Exception handler called!");
    }
}
```

In the example above, the exception filter is the function MyExceptionFilter and the handler is the code that simply prints that it was called. When registering a vectored exception handler, the handler function must be of the prototype `typedef LONG(NTAPI* ExceptionHandler_t)(PEXCEPTION_POINTERS ExceptionInfo)`.

It turns out that the prototype for exception filters is actually compatible with the prototype above. What does this mean? We can reuse the same exception handler we wrote for the VEH implementation of Exception Oriented Programming by using it as an exception filter.

```
void my_bad_code() {
    __try {
        __int3;
    } __except(VectoredExceptionHandler(GetExceptionInformation())) {
        printf("Exception handler called!");
    }
}
```

In the code above, the vectored exception handler is invoked using the GetExceptionInformation macro, which provides the function the exception information structure it can both read and modify.

That's all that you need to do to get Exception Oriented Programming working with standard SEH! Besides ensuring that the instruction locations found are compatible, the vectored exception handler is directly compatible when used as an exception filter.

Why is standard SEH significantly better than using VEH for Exception Oriented Programming? SEH is built into the binary itself and is used legitimately *everywhere*. Unlike vectored exception handling, there is no global function to register your handler.

From the perspective of *static* detection, there are practically no indicators that a given SEH handler is used for Exception Oriented Programming. Although dynamic detection may be possible, it is significantly harder to implement compared to if you were using Vectored Exception Handlers.

Bypassing the macOS Hardened Runtime

Up to this point, the examples around abuse of the method have been largely around the Windows operating system. In this section, we will discuss how we can abuse Exception Oriented Programming to bypass security mitigations on macOS, specifically parts of the Hardened Runtime.

The macOS Hardened Runtime is intended to provide "runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering".

One security mitigation imposed by the Hardened Runtime is the restriction of just-in-time (JIT) compilation. For app developers, these restrictions can be bypassed by adding entitlements to disable certain protections.

The `com.apple.security.cs.allow-jit` entitlement allows an application to allocate writable/executable (WX) pages by using the `MAP_JIT` flag. A second alternative, the `com.apple.security.cs.allow-unsigned-executable-memory` entitlement, allows the

application to allocate WX pages without the need of the `MAP_JIT` flag. With Exception Oriented Programming however, an attacker can execute just-in-time shellcode without needing any entitlements.

The flow for the preparation stage of this implementation is as follows:

1. The application will register a `SIGTRAP` signal handler using `sigaction` and the `SA_SIGINFO` flag.
2. The application will split each instruction of the given shellcode using any disassembler. For the example proof-of-concept, we will use the Zydis disassembler library.
3. For each split instruction, the application will attempt to find an instance of that instruction present in the executable memory of any loaded modules. Executable memory regions can be recursively enumerated using the `mach_vm_region_recurse` function. These memory locations will be stored for later use by the signal handler.
4. The application will finish preparing by finding any instance of an `int3` instruction (a single 0xCC byte). This instruction will be stored for use by the signal handler and is returned to the caller which will invoke the arbitrary shellcode.

Once the necessary memory locations have been found, the caller can invoke the arbitrary shellcode by executing the `int3` instruction that was returned to them.

1. Once the caller has invoked the `int3` instruction, the signal handler will be called. The signal handler should determine if this exception is for executing arbitrary shellcode by comparing the fault address - 1 with the previously stored location of the `int3` instruction. One must be subtracted from the fault address because in the `SIGTRAP` signal handler, the fault address points to the instruction pointer whereas we need the instruction that caused the exception.
2. If the breakpoint is indeed for the arbitrary shellcode, then the signal handler should:
    1. Retrieve the list of legitimate instructions needed to simulate the arbitrary shellcode.
    2. Store these instructions in thread-local storage.
    3. Set the instruction pointer to the first legitimate instruction to execute.
    4. Set the Trap flag on the FLAGS register.
    5. Continue execution.

3. The rest of the instructions will call the signal handler, however, unlike Vectored Exception handlers, there is no error code passed differentiating a breakpoint and a single step exception. The signal handler can determine if the exception is for a legitimate instruction being executed by checking its thread-local storage for the previously set context. In these cases, the signal handler should:
    1. Retrieve the list of legitimate instructions to execute from the thread-local storage.
    2. Set the instruction pointer to the next legitimate instruction's memory location.
    3. If this instruction is *not* the last instruction to execute, set the Trap flag on the FLAGS register. Otherwise, do *not* set the Trap flag.
    4. Continue execution.

Assuming the shellcode ends with a return instruction, eventually the execution flow will be gracefully returned to the caller.

Exception Oriented Programming highlights a fundamental design flaw with the JIT restrictions present in the Hardened Runtime. The JIT mitigation assumes that to execute code "just-in-time", an attacker must have access to a WX page. In reality, an attacker can abuse a large amount of the instructions already present in legitimate modules to execute their own malicious shellcode.

Proof of Concept

Both the Windows and macOS proof-of-concept utilities can be accessed at this repository.

Conclusion

As seen with the new methodology in this article, code execution can be achieved without the need of dedicated memory for that code. When considering future research into runtime code execution, it is more effective to look at execution from a high-level perspective, an objective of executing the operations in a piece of code, instead of focusing on the requirements of existing methodology.

In part 2 of this series, we will explore how Exception Oriented Programming expands the possibilities for buffer overflow exploitation on Windows. We'll explore how to evade Microsoft's ROP mitigations such as security cookies and SafeSEH for gaining code execution from common vulnerabilities. Make sure to follow my Twitter to be amongst the first to know when this article has been published!

Parallel Discovery

Recently, another researcher (@x86matthew) published an article describing a similar idea to Exception Oriented Programming, implemented using vectored exception handlers for x86.

Whenever my research leads me to some new methodology I consider innovative, one practice I take is to publish a SHA256 hash of the idea, such that in the future, I can prove that I discovered a certain idea at a certain point in time. Fortunately, I followed this practice for Exception Oriented Programming.

On February 3rd, 2021, I created a public gist of the follow SHA256 hash:

```
5169c2b0b13a9b713b3d388e61eb007672e2377afd53720a61231491a4b627f7
```

To prove that this hash is a representation of a message summarizing Exception Oriented Programming, here is the message you can take a SHA256 hash of and compare to the published one above.

> Instead of allocating executable memory to execute shellcode, split the shellcode into individual instructions, find modules in memory that have the instruction bytes in an executable section, then single step over those instructions (changing the RIP to the next instruction and so on).

Since the core idea was published by Matthew, I wanted to share my additional research in this article around stealthier SEH exception handlers and how the impact is not only limited to Windows. In a future article, I plan on sharing my additional research on how this methodology can be applied on previously unexploitable buffer overflow vulnerabilities on Windows.