Reverse Engineering

1. Introduction to Reverse Engineering, Stack Frame Analysis

Ing. Tomáš Zahradnický, EUR ING, Ph.D.
Ing. Josef Kokeš

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Czech Technical University in Prague
Faculty of Information Technology
Department of Information Security

Version 2021-09-23

Table of Contents I

Table of Contents II

Reverse Engineering

Software Engineering [1]

Software engineering is the study and application of engineering to the design, development, and maintenance of software.

- In software engineering, we start with source code, which is then compiled into object code and linked into executable code.
- Software reverse engineering reverses this process.

(Software) Reverse Engineering (RE) [2]

Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.

- During this process we document and try to understand how the studied system works.

Reverse Engineering Uses

RE is used to study how a software product works in order to:

- find which algoritms, methods, file formats, and protocols the product uses;
- find vulnerabilities in the product;
- achieve understanding of the product as the first step to designing a competitive product;
- design a product cooperating with the product;
- decide whether the product is malicious or not.

The above steps have two aspects: **ethical** and **legal**.

Ethical Aspects of Reverse Engineering

- Ethical RE uses compatible with Czech law, assuming a valid license:
  - create interconnections with a proprietary product;
  - discover (private) interfaces to extend the functionality of a proprietary product;
  - fix vulnerabilities in a proprietary product.

- Unethical uses occur when RE is used to:
  - bypass copy protection or a DRM;
  - discover a product's serial number scheme and create a keygen;
  - create a competitive product.

Legal Aspects of Reverse Engineering

RE is often denied by the End User License Agreement (EULA).

You may not reverse engineer, decompile, or disassemble the Software, except and only to the extent that such activity is expressly permitted by this EULA or applicable law notwithstanding this limitation. [Microsoft EULA]

Czech Author's Act no. 121/2000 Coll., as amended, says in § 66:

(1) Do práva autorského nezasahuje oprávněný uživatel rozmnoženiny počítačového programu, jestliže:

a) rozmnožuje, překládá, zpracovává, upravuje či jinak mění počítačový program, je-li to nezbytné k využití oprávněně nabyté rozmnoženiny počítačového programu, činí-li tak při zavedení a provozu počítačového programu nebo opravuje-li chyby počítačového programu,

b) jinak rozmnožuje, překládá, zpracovává, upravuje či jinak mění počítačový program, je-li to nezbytné k využití oprávněně nabyté rozmnoženiny počítačového programu v souladu s jeho určením, není-li dohodnuto jinak,

d) zkoumá, studuje nebo zkouší sám nebo jím pověřená osoba funkčnost počítačového programu za účelem zjištění myšlenek a principů, na nichž je založen kterýkoli prvek počítačového programu, činí-li tak při takovém zavedení, uložení počítačového programu do paměti počítače nebo při jeho zobrazení, provozu či přenosu, k němuž je oprávněn,

e) rozmnožuje kód nebo překládá jeho formu při rozmnožování počítačového programu nebo při jeho překladu či jiném zpracování, úpravě či jiné změně, je-li k ní oprávněn, a to samostatně nebo prostřednictvím jím pověřené osoby, jsou-li takové rozmnožování nebo překlad nezbytné k získání informací potřebných k dosažení vzájemného funkčního propojení nezávisle vytvořeného počítačového programu s jinými počítačovými programy, jestliže informace potřebné k dosažení vzájemného funkčního propojení nejsou pro takové osoby dříve jinak snadno a rychle dostupné a tato činnost se omezuje na ty části počítačového programu, které jsou potřebné k dosažení vzájemného funkčního propojení.

(4) Informace získané při činnosti podle odstavce 1 písm. e) nesmějí být poskytnuty jiným osobám, ledaže je to nezbytné k dosažení vzájemného funkčního propojení nezávisle vytvořeného počítačového programu, ani využity k jiným účelům než k dosažení vzájemného funkčního propojení nezávisle vytvořeného počítačového programu. Dále nesmějí být tyto informace využity ani k vývoji, zhotovení nebo k obchodnímu využití počítačového programu podobného tomuto počítačovému programu v jeho vyjádření nebo k jinému jednání ohrožujícímu nebo porušujícímu právo autorské.

→ We can legally RE software to understand how it works and to create interconnections with it. What EULAs say about RE does not need to be relevant! (a law has a priority over EULA). But beware of article 1 letter b!

Amount of Information

Source Code	**Object Code**	**Executable**
source code	debug info	–
comments	(comments)	–
source files separated	object files separated	linked together
libraries standalone	libraries standalone	merged

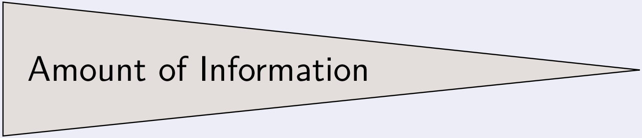

Table: Amount of information in source, object, and executable code.

- Object code contains less information than source code.
- Executable contains even less information than object code.

→ Going back is a difficult task because of the lack of information!

Obtaining Source Code from an Executable

- Due to the lack of information in executables, it is impossible to reconstruct their source code in full. Though we try to get as close as possible, the recovered source code is generally not compilable. Round-trips source-executable-source-executable are unrealistic.
- We can use:
  - **disassembling** — a transformation of executable code from a machine code into the assembly language of the target processor. We end up with a huge amount of code, human beings need a higher-level programming language. Disassembly may be either performed on a dead code or by running a live code in a debugger. Disassembly can be complicated by means of obfuscation (e.g. opaque predicates, API calls obfuscation, ...), encoding, packing, etc.
  - **decompilation** — a transformation of executable code into a higher-level programming language (usually C, Python, or Perl). Not 100 percent, no round trips possible. Obfuscation, packers [Lec 4], and code polymorphism may make the situation very difficult.

Dead Code Analysis

Dead Code Analysis (aka. Static Analysis)

Dead Code Analysis is an analysis performed on a non-running executable code with a goal to study and document the code's behavior. This is typically done by the means of a disassembler.

1. An executable is disassembled [Lec 4].
2. Should the disassembled result be studied now?

Dead Code Analysis

Dead Code Analysis (aka. Static Analysis)

Dead Code Analysis is an analysis performed on a non-running executable code with a goal to study and document the code's behavior. This is typically done by the means of a disassembler.

1. An executable is disassembled [Lec 4].
2. Should the disassembled result be studied now?
3. With so many lines of code (LOC)? Reduce them!

Dead Code Analysis

Dead Code Analysis (aka. Static Analysis)

Dead Code Analysis is an analysis performed on a non-running executable code with a goal to study and document the code's behavior. This is typically done by the means of a disassembler.

1. An executable is disassembled [Lec 4].
2. Should the disassembled result be studied now?
3. With so many lines of code (LOC)? Reduce them!
4. Reduce LOCs by identifying the compiler [Lec 5], statically linked library code [Lec 5] and data and drop them!

Dead Code Analysis

Dead Code Analysis (aka. Static Analysis)

Dead Code Analysis is an analysis performed on a non-running executable code with a goal to study and document the code's behavior. This is typically done by the means of a disassembler.

1. An executable is disassembled [Lec 4].
2. Should the disassembled result be studied now?
3. With so many lines of code (LOC)? Reduce them!
4. Reduce LOCs by identifying the compiler [Lec 5], statically linked library code [Lec 5] and data and drop them!
5. Extract all possible information left in the executable and annotate the assembly with it (code/data/bss segments, extract class names, strings, RTTI information, exception information, stack frames, parameter type information from known API calls, etc.)
6. Ultimately let a human being study the leftover code!

Dead Code Analysis

Dead Code Analysis (aka. Static Analysis)

Dead Code Analysis is an analysis performed on a non-running executable code with a goal to study and document the code's behavior. This is typically done by the means of a disassembler.

1. An executable is disassembled [Lec 4].
2. Should the disassembled result be studied now?
3. With so many lines of code (LOC)? Reduce them!
4. Reduce LOCs by identifying the compiler [Lec 5], statically linked library code [Lec 5] and data and drop them!
5. Extract all possible information left in the executable and annotate the assembly with it (code/data/bss segments, extract class names, strings, RTTI information, exception information, stack frames, parameter type information from known API calls, etc.)
6. Ultimately let a human being study the leftover code!
7. Even the disassembler may get confused [Lec 4]!

Live Code Analysis

Live Code Analysis (aka. Dynamic Analysis)

Live Code Analysis is an analysis performed on a running target with a goal to study and document the target's code behavior. This is typically done by the means of a debugger.

1. An executable is typically loaded into a debugger and studied there.
2. Breakpoints and watchpoints can be set and the actual values of registers and memory can be studied at run time.
3. Software can resist debugging (using killer threads, detecting a debugger, denying attaching, etc.), use CRCs to check for software breakpoints, ... [Lecture 6]

Application Binary Interfaces/Calling Conventions

Application Binary Interfaces (also called Calling Conventions) [3] is a document describing how should a binary code behave on a target platform in order to be compatible with 3rd party binary code and the operating system. This includes how:

- the parameters are passed from function to function;
- are stack and data aligned;
- CPU registers are used (which are volatile, which need to be preserved);
- are symbol names mangled;
- structs/classes are constructed in memory;
- virtual functions in object oriented languages are called;
- run time type identification (RTTI) is performed;
- exceptions are handled;
- ...

Data Alignment I

Access to unaligned data is slower (BIE-APS) or even impossible on some CPUs (e.g. MC68000 cannot read a long word [4B] from an odd address) → data must be aligned there. Current processors support unaligned data, but access is slower → align data wherever possible.

	i386		x86_64	
data type	**MSVC**	**GCC**	**MSVC**	**GCC**
1 byte char	1	1	1	1
2 byte char	4	2	4	2
4 byte char	4	4	4	4
8 byte char	8	8	8	8
float	4	4	4	4
double	8	8	8	8
pointer	4	4	8	8

Table: Static data alignment for MSVC and GCC 3.x compilers [3].

Data Alignment II

	i386		x86_64	
data type	**MSVC**	**GCC**	**MSVC**	**GCC**
1 byte char	1	1	1	1
2 byte char	4	2	2	2
4 byte char	4	4	4	4
8 byte char	8	8	4/8	8
float	4	4	4	4
double	8	8	8	8
pointer	4	4	8	8

Table: Structure/class members alignment for MSVC and GCC 3.x compilers [3].

- Alignment can cause problems when using different compilers.
- For this reason we may need to control the alignment.
- In assembly language the `.align num_bytes` pseudo instruction is used.
- C/C++ provides a `#pragma pack` to control the alignment:

Data Alignment III

```
#pragma pack(push)
// Store the current alignment
#pragma pack(1)
  // Everything from here on aligned on a 1 byte boundary
  typedef struct _MYSTRUCT {
    char f_Field1;     // The f_Filler1 field is used to
    char f_Filler1[1]; // align f_Int at a 2 byte boundary
    int  f_Int;        // Aligned at a 2-byte boundary!
  } MYSTRUCT;
#pragma pack(pop)
// Former alignment restored
```

Stack Alignment I

Stack may also need to be aligned. Traditionally, 32-bit systems align stack on a 4-byte boundary while 64-bit systems align on a 16-byte boundary. The alignment breaks when a call instruction is executed causing stack to point to an address $A \equiv 12 \pmod{16}$ (when aligned to 16 bytes in the 32-bit mode) or $A \equiv 8 \pmod{16}$ in the 64-bit mode.

```
8048528 <main>:
8048528:  55                        push   %ebp
8048529:  89 e5                     mov    %esp,%ebp
804852b:  83 e4 f0                  and    $0xfffffff0,%esp    // Realign the stack
804852e:  83 ec 20                  sub    $0x20,%esp
8048531:  c7 44 24 1c 00 00 00 00   movl   $0x0,0x1c(%esp)
8048539:  c7 44 24 08 03 00 00 00   movl   $0x3,0x8(%esp)      // 'Push' the 3rd parameter
8048541:  c7 44 24 04 02 00 00 00   movl   $0x2,0x4(%esp)      // 'Push' the 2nd parameter
8048549:  c7 04 24 01 00 00 00      movl   $0x1,(%esp)         // 'Push' the 1st parameter
8048550:  e8 76 ff ff ff            call   80484cb <f>
8048555:  89 44 24 1c               mov    %eax,0x1c(%esp)
8048559:  8b 44 24 1c               mov    0x1c(%esp),%eax
804855d:  89 44 24 04               mov    %eax,0x4(%esp)
8048561:  c7 04 24 10 86 04 08      movl   $0x8048610,(%esp)
8048568:  e8 23 fe ff ff            call   8048390 <printf@plt>
804856d:  b8 00 00 00 00            mov    $0x0,%eax
8048572:  c9                        leave
8048573:  c3                        ret                        // The function ends at 8048573
```

Stack Alignment II

```
8048574:   66 90                xchg   %ax,%ax        // The following no-ops are used
8048576:   66 90                xchg   %ax,%ax        // as a filler so that the next
8048578:   66 90                xchg   %ax,%ax        // function can begin at a 16-byte
804857a:   66 90                xchg   %ax,%ax        // boundary 8048580.
804857c:   66 90                xchg   %ax,%ax
804857e:   66 90                xchg   %ax,%ax
```

Name Mangling I

Some languages allow function/operator/method overloading and therefore the symbol name alone is not sufficient to distinguish between two or more overloaded symbols. For this reason additional information must be encoded into the symbol's name. The process of encoding this additional information is called **name mangling** (Microsoft calls this **name decoration**). In C++, the following information is usually encoded:

- symbol's name;
- all symbol's namespaces;
- whether the symbol is `const` and/or `volatile`;
- whether the symbol is `public`, `protected`, or `private`;
- if the symbol is a function or a method, then its arguments (some compilers encode the return value as well);
- whether the symbol is a scalar or a vector data type.

Mangled names provide a wealth of information.

Name Mangling II

Let us inspect the following mangled name:

`?_GetOutputTechnologyString@CDisplayData@@AAEXPAGHH@Z`

This is a Microsoft decorated name. These names always start with `?` followed with the method's name `_GetOutputTechnologyString`. We see that the method is a part of the `CDisplayData` class (or struct). We can also observe that `@` serves as a delimiter. Next there's the `AAEXPAGHH` part and according to [3] we get:

`A` `private`	`A` neither `const` nor `volatile`	`E` `__thiscall`	`X` returns `void`	`P` pointer to
`A` neither `c` nor `v`	`G` `unsigned short`	`H` `int`	`H` `int`	`@Z` end

Table: Demangling a symbol's name.

Name Mangling III

Ultimately we obtain the function's prototype:

```
private __thiscall void CDisplayData::_GetOutputTechnologyString(
  CDisplayData* this, unsigned short*, int, int
);
```

Instance methods receive their `this` pointer automatically as the first (implicit) parameter. On Windows in the 32-bit mode, the `__thiscall` convention is used and the `this` pointer is passed in `ECX`. In the 64-bit mode the `this` pointer is passed as the first (implicit) parameter.

The real function prototype would be:

```
private void CDisplayData::_GetOutputTechnologyString( LPWSTR, int, int );
```

Note

Methods declared with the `static` keyword (class methods) **do not** receive the `this` pointer as the first parameter!

Register Use

ABI also defines how are the CPU's registers used: which of them are volatile (the callee can trash the register's value), which register serves as the stack pointer, which is the frame pointer, where are the parameters passed from function to function, and where the result is returned.

Use	Win32	32b Linux/BSD/OS X	Win64	64b Linux/BSD/OS X
volatile	`EAX, ECX, EDX, ST(0)-ST(7)`	`EAX, ECX, EDX, ST(0)-ST(7)`	`RAX, RCX, RDX, R8-R11, ST(0)-ST(7)`	`RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7)`
callee-saves	`EBX, ESI, EDI, EBP`	`EBX, ESI, EDI, EBP`	`RBX, RSI, RDI, RBP, R12-R15`	`RBX, RBP, R12-R15`
parameter transfer	special	special	`RCX, RDX, R8, R9`	`RDI, RSI, RDX, RCX, R8, R9`
return values	`EAX, EDX, ST(0), XMM0, YMM0, ZMM0`	`EAX, ST(0), XMM0, YMM0, ZMM0`	`RAX, ST(0), XMM0, YMM0, ZMM0`	`RAX, RDX, ST(0), XMM0, YMM0, ZMM0`

Table: Register usage [3] (simplified).

Calling Conventions I

Calling conventions [3] specify how functions are called, who and where puts their parameters, where the result is returned, and who performs the parameter cleanup, whether it is the caller or the callee (function). The `__cdecl` calling convention is implicit for all C functions, while the `__thiscall` is implicit for methods.

Convention	Par. in regs.	Ordering	Cleanup done by
`__cdecl`	—	C	caller
`__stdcall`	—	C	function
`__pascal`	—	Pascal	function
GNU	—	C	hybrid
`__fastcall`	ECX, EDX	C	function
`__thiscall`	ECX	C	function

Table: Calling conventions for the 32-bit architecture.

Calling Conventions II

The 64-bit architecture calling conventions are far simpler:

Convention	Par. in regs.	Ordering	Cleanup done by
Windows	`RCX`, `RDX`, `R8`, `R9`	C	caller
Linux/BSD/OS X	`RDI`, `RSI`		
	`RDX`, `RCX`, `R8`, `R9`	C	function

Table: Calling conventions for the 64-bit architecture.

Calling Conventions III

The convention is always a part of the function/method's declaration:

```
BOOL WINAPI MessageBeep( UINT uType );
```

is really:

```
BOOL __stdcall MessageBeep( UINT uType );
```

and a pointer to this function must contain the convention too:

```
BOOL (__stdcall *pfnMB)(UINT) = (BOOL (__stdcall*)(UINT))GetProcAddress(...);
```

Note

Omission or use of an improper calling convention results in a severe bug.

Let's start

So far we have discussed:

- what is the Reverse Engineering of computer software;
- what are the Application Binary Interfaces;
- what are the Calling Conventions;
- why do we need Name Mangling.

Now, let's start with the real work — analyze a real function[1]! We will:

- analyze the function's stack frame;
- analyze conditional jump instructions to find sign of stack frame items;
- analyze function calls to rectify data types;
- reconstruct local variables of the function.

[1] if you are uncertain in the assembly language, register also for BIE-SOJ!

Reverse Engineering a Function

Each function consists of 3 parts:

1. prologue (optional),
2. body, and
3. epilogue (optional).

Prologue are the first several lines of each function creating the stack frame, allocating space for local variables, optionally aligning the stack, and saving all non-volatile registers used by the function. Optionally the stack canary is inserted, as well as the `EXCEPTION_REGISTRATION` structure, including MSVC specific extensions, if the structured exception handling (Windows) is used.

Epilogue are the ending lines of a function just before the `ret` instruction, that dispose of the stack frame. If the stack canary got inserted in the prologue, it is checked here, and if there's mismatch, the program is aborted.

In RE, we can move over the prologue quickly to focus on the body.

Prologue I

When a function is entered, the return address is at the top of the stack. The ESP register points to the return address. Then the prologue starts:

A typical prologue (MSVC/IA-32)

```
push   %ebp          // EBP is non-volatile, must be saved!
mov    %esp, %ebp    // Items on the stack will be referred to relatively to EBP
sub    $0x20, %esp   // Create a 32 B long space in the stack for locals
push   %ebx          // Optionally push EBX if the function overwrites it
push   %esi          // Optionally push ESI if the function overwrites it
push   %edi          // Optionally push EDI if the function overwrites it
```

Now EBP will be used to refer to items on the stack so:

- `EBP-20...EBP-1` refers to local variables;
- `EBP+ 0...EBP+3` refers to the saved EBP value;
- `EBP+ 4...EBP+7` refers to the return address;
- `EBP+ 8...EBP+B` refers to the first 4-byte argument if present;
- `EBP+ C...EBP+F` refers to the second 4-byte argument if present;

`EBP+offset`$\geq$`8` is a parameter, while `EBP-offset` is a local variable.

Prologue II

The x86 architecture has the `enter` instruction which is supposed to create a stack frame. These two prologues are functionally equivalent. The μops are valid for the Intel Haswell architecture [3]:

A default prologue on the IA-32 architecture

```
push    %ebp           // 2 μops
mov     %esp, %ebp     // 1 μop
sub     $0x20, %esp    // 1 μop
```

A prologue with the `enter` instruction

```
enter   $0x20, $0x0     // 12 μops
```

Note

Since the `enter` instruction is significantly slower than the `push/mov/sub` combo, it is rarely used to create a stack frame.

Stack Canaries

In order to make stack smashing (an intentional overwriting of data on the stack as a result of a specially crafted input to the program) very difficult, a random word called **a canary** is inserted on the stack as a part of the prologue code. Before the function exits, the value of the canary is checked and if a mismatch is found, the program is not allowed to return from the current function and is immediately terminated. Termination is necessary, as the attacker could have overwritten the return address and hijacked the program to execute her code.

The **canary** is generated at random by the runtime at program's startup and may even be different for each function in the program — or may be the same for all functions. This is implementation specific.

Stack Canaries — GNU

Linux **Stack Protector** (`-fstack-protector`) inserts a word saved at `[gs:0x14]` as follows:

Canary Insertion

```
mov %gs:0x14, %eax
mov %eax, -0xc(%ebp)
```

Canary Check

```
  mov     -0xc(%ebp), %ecx
  xor     %gs:0x14, %ecx
  je      canary_check_ok
  call    <__stack_chk_fail@plt>
canary_check_ok:
  // Continue here
```

Stack Canaries — MSVC Windows I

Windows calls stack protection mechanism a **Buffer Security Check**. The idea is to generate a global `DWORD __security_cookie` at the start of the program randomly (see the next slide), and then xor it with `EBP` and insert it at the stack in the prologue, and check whether it remained unchanged in the epilogue.

Canary Insertion

```
// Load cookie into EAX
mov  eax, [___security_cookie]

// XOR with EBP calculates the canary
xor  eax, ebp

// Put the cookie on the stack
mov  [ebp-4], eax
```

Canary Check

```
// Load the canary into ECX
mov  ecx, [ebp-4]

// Recalculate __security_cookie
xor  ecx, ebp

// Abort if the result mismatches
call j_@__security_check_cookie@4
```

Stack Canaries — MSVC Windows II

```
unsigned int __security_init_cookie() {
  unsigned int  result;
  LARGE_INTEGER perfctr;
  unsigned int  cookie;
  FILETIME      systime;

  systime.ft_scalar = 0;
  if( __security_cookie != 0xBB40E64E // Is it the initial value?
    && (result = __security_cookie & 0xFFFF0000) != 0 )
        __security_cookie_complement = ~__security_cookie;
  else {
    GetSystemTimeAsFileTime((LPFILETIME)&systime);
    cookie = systime.ft_struct.dwHighDateTime;
    cookie ^= systime.ft_struct.dwLowDateTime;
    cookie ^= GetCurrentProcessId();
    cookie ^= GetCurrentThreadId();
    cookie ^= GetTickCount();
    QueryPerformanceCounter(&perfctr);
    result = perfctr.LowPart ^cookie;
    cookie ^= perfctr.LowPart;
    cookie ^= perfctr.HighPart;
    if ( cookie == 0xBB40E64E ) // Cookie == initial?
```

Stack Canaries — MSVC Windows III

```
      cookie = 0xBB40E64F;
    else if ( !(cookie & 0xFFFF0000) ) {
      result = cookie | (cookie << 16);
      cookie |= cookie << 16;
    }

    __security_cookie = cookie;
    __security_cookie_complement = ~cookie;
  }

  return result;
}
```

Structured Exception Handling (32-bit) I

Structured Exception Handling (SEH) is specific to Windows. MSVC C SEH (`__try`/`__except`/`__finally`) and C++ exceptions (`try`/`catch`) are built on top of SEH. When a system-level exception occurs (such as division by zero or a NULL pointer dereference) an exception handler is called. The handler is defined in the following structure (found on the stack):

```
struct EXCEPTION_REGISTRATION {
  EXCEPTION_REGISTRATION* prev;     // Pointer to the previous E. R.
  LPVOID                  handler;  // Pointer to an exception handler
};
```

Pointer to the topmost `EXCEPTION_REGISTRATION` is found in the first field (`FS:[0]`) of the Thread Information Block (TIB), which is pointed to by the `FS` register.

Structured Exception Handling (32-bit) II

When an exception occurs, the OS first calls the handler found in the `handler` field of the topmost `EXCEPTION_REGISTRATION`. The handler decides whether it wants to handle the exception. If the handler refuses to handle the exception, the OS goes to the previous (superior) `EXCEPTION_REGISTRATION`, which is pointed to by the `prev` field, and calls its handler. This is repeated until one of the handlers in the chain chooses to handle the exception, or until the head of the list is hit and the OS's "last resort" handler is used to terminate the program or attach a debugger if one's present.

Installing a raw `EXCEPTION_REGISTRATION`

```
push ebp                  // Standard prologue start
mov  ebp, esp             // Establish the base pointer
push MyExceptionHandler   // Push EXCEPTION_REGISTRATION.handler
push fs:[0]               // Push EXCEPTION_REGISTRATION.prev
mov  fs:[0], esp          // Set the top EXCEPTION_REGISTRATION
```

Structured Exception Handling (32-bit) III

Removing an EXCEPTION_REGISTRATION

```
mov eax, [ebp-8]     // Load the prev EXCEPTION_REGISTRATION into EAX
mov fs:[0], eax      // Set the prev EXCEPTION_REGISTRATION as the topmost one
```

The exception handler has the following prototype:

```
EXCEPTION_DISPOSITION __cdecl except_handler(
  struct _EXCEPTION_RECORD* ExceptionRecord,
  void* EstablisherFrame,
  struct _CONTEXT *ContextRecord,
  void* DispatcherContext
);
```

The `ExceptionRecord` contains information about the exception such as the exception code, address, flags, etc. The second parameter, the `EstablisherFrame`, points at the `EXCEPTION_REGISTRATION` in the frame where the exception occurred. `ContextRecord` contains the state of the thread.

Structured Exception Handling (32-bit) IV

Once a suitable handler is found, the handler chain is called again up to the same point, except with `ExceptionRecord->ExceptionFlags` with the `EH_UNWINDING` flag set. This means that the stack is going to be unwound to the previous frame up to the frame of the suitable handler. At this time any cleanup code should be performed (in C++ destructors are called, statements in the `__finally` block executed, etc.).

Since this is a lot of work, MSVC runtime (not the operating system!) provides `_except_handler3` and the newer `_except_handler4` functions which do most of the work for you! The `EXCEPTION_REGISTRATION` structure then contains a pointer to the MSVC handler `_except_handler3/4` and the `EXCEPTION_REGISTRATION` structure is extended by 2 additional fields (a try level and scope table pointer) allowing to handle nested `__try` blocks with a single handler.[6]

Structured Exception Handling (32-bit) V

```
struct EH3_EXCEPTION_REGISTRATION {
  EH3_EXCEPTION_REGISTRATION* prev;       // Pointer to the previous E. R.
  LPVOID                      handler;    // Pointer to the exception handler
  SCOPETABLE_ENTRY*           ScopeTable; // Pointer to the scope table
  DWORD                       TryLevel;   // Current try block identifier
};
```

The scope table typically resides in a read-only section of the memory and, in the basic case (SEH3), consists of a list of records:

```
struct _SCOPETABLE_ENTRY {
  DWORD   EnclosingLevel;     // Identifier of the superior try block
  LPVOID* FilterFunc;         // Filter function
  LPVOID* HandlerFunc;        // Handler function
};
```

SEH4 adds more functionality, such as masking of the scope table pointer with the security cookie.

Structured Exception Handling (32-bit) VI

In this case the whole function only has one SEH exception registration record. To enter a new `try` block it is sufficient to write its identifier into the `TryLevel` variable. When processing an exception, the handler reads the value of `TryLevel` and the address of the scope table. `TryLevel` is used as an index into the table, identifying one record. The handler calls the `FilterFunc` function which returns information whether the record is interested in the current exception. If it is, `HandlerFunc` is called to handle the exception and `TryLevel` is set to `EnclosingLevel`. In the other case we use `EnclosingLevel` as an index of the parent block, which is then asked to handle the exception.

The top level record of the scope table has index -2 (`__try` of C) or -1 (`try` of C++). If even the top level record fails to resolve the exception, the SEH3 handler ends and returns control to the standard SEH of the OS.

The complete setup then looks like this:

Structured Exception Handling (32-bit) VII

```
00412AB0  push   ebp
00412AB1  mov    ebp,esp
00412AB3  push   0FFFFFFFEh                      // Current try level is -2
00412AB5  push   418540h                         // Push pointer to the scope table
00412ABA  push   411069h                         // Push __except_handler4 address
00412ABF  mov    eax,dword ptr fs:[00000000h]   // Copy the top EXC_REG from TIB
00412AC5  push   eax                             // Push link to the previous EXC_REG
00412AC6  sub    esp,10h
00412ACC  mov    eax,dword ptr ds:[__security_cookie]
00412AD1  xor    dword ptr [ebp-8],eax           // Obfuscate the scope table pointer
00412AD4  xor    eax,ebp
00412AD6  mov    dword ptr [ebp-1Ch],eax         // Canary
00412AD9  push   ebx
00412ADA  push   esi
00412ADB  push   edi
00412ADC  push   eax
00412ADD  lea    eax,[ebp-10h]                   // EAX = & EXCEPTION_REGISTRATION
00412AE0  mov    dword ptr fs:[00000000h],eax   // Set top EXC_REG in TIB
00412AE6  mov    dword ptr [ebp-18h],esp         // Save stack top
```

Structured Exception Handling (32-bit) VIII

```
// The __try block starts here
00412C74 mov  dword ptr [ebp-4],0          // Enter a new try block
00412C7B mov  dword ptr ds:[0],0           // Dereference a NULL pointer - boom!
00412C85 mov  dword ptr [ebp-4],0FFFFFFFEh // Revert tryLevel
00412C8C jmp  $LN6+18h (0412CC1h)          // Exit __try, continue after the block

// Code for calling the exception filter
// MyExceptionFilter( GetExceptionCode(), GetExceptionInformation())
00412C8E mov  eax,dword ptr [ebp-14h]      // GetExceptionInformation()
00412C91 mov  ecx,dword ptr [eax]          //-> ExceptionRecord
00412C93 mov  edx,dword ptr [ecx]          //-> ExceptionCode
00412C95 mov  dword ptr [ebp-5Ch],edx      // Store the exc.  code into a temp var
00412C98 mov  eax,dword ptr [ebp-14h]      // GetExceptionInformation()
00412C9B push eax                          // Push it onto the stack
00412C9C mov  ecx,dword ptr [ebp-5Ch]      // GetExceptionCode()
00412C9F push ecx                          // Push it onto the stack
00412CA0 call MyExceptionFilter (0411104h) // Call the filter
00412CA5 add  esp,8                        // Pop the parameters from the stack
00412CA8 ret                               // Return to the exception handler
```

Structured Exception Handling (32-bit) IX

```
// The __except block starts here
00412CA9 mov  esp,dword ptr [ebp-18h]       // Set the stack top
00412CAC push 417A50h                       // Push parameters for printf
00412CB1 call dword ptr ds:[41A120h]        // Call printf
00412CB7 add  esp,4                         // Clean the parameters
00412CBA mov  dword ptr [ebp-4],0FFFFFFFEh  // Return to the top level try block

// Restore the top EXCEPTION_REGISTRATION
00412CC1 mov  ecx,dword ptr [ebp-10h]       // Load EXC_REG.prev to ECX
00412CC4 mov  dword ptr fs:[0],ecx          // Set the topmost EXC_REG in TIB

// Standard epilogue
00412CCB pop  ecx
00412CCC pop  edi
00412CCD pop  esi
00412CCE pop  ebx
00412CCF mov  esp,ebp
00412CD1 pop  ebp
00412CD2 ret
```

Who Installs the Fallback Handler? I

You might have noticed that there's an OS handler which decides to handle every exception by terminating the program. Where does it get installed from? Let's look at the call stack when a new thread is created:

```
application!ThreadRoutine  // This is our thread's proc
  kernel32!BaseThreadInitThunk+0xe
    ntdll!__RtlUserThreadStart+0x70
      ntdll!_RtlUserThreadStart+0x1b
```

```
kernel32!BaseThreadInitThunk:
  761cee0a  mov   edi,edi
  761cee0c  push  ebp
  761cee0d  mov   ebp,esp
  761cee0f  test  ecx,ecx
  761cee11  jne   kernel32!BaseThreadInitThunk+0x15 (761cef64)
  761cee17  push  dword ptr [ebp+8]    // Push thread arg.
  761cee1a  call  edx                  // Call thread proc
  761cee1c  push  eax                  // Push the thread result
  761cee1d  call  dword ptr [kernel32!_imp__RtlExitUserThread (7618170c)]
  // Unreachable
```

Obviously `BaseThreadInitThunk` does not install the handler!

Who Installs the Fallback Handler? II

Let's go up in the call stack and examine `__RtlUserThreadStart`:

```
ntdll!__RtlUserThreadStart:
  77f237c4  push  14h
  77f237c6  push  offset stru_77f11278 // Contains info about filter and handler
  77f237cb  call  __SEH_prolog4
  77f237d0  and   [ebp+ms_exc.registration.TryLevel], 0 // Entering the __try block
  77f237d4  mov   eax, _Kernel32ThreadInitThunkFunction
  77f237d9  push  [ebp+c]
  77f237dc  test  eax, eax
  77f237de  jz    loc_77ec5e77
  77f237e4  mov   edx, [ebp+8]
  77f237e7  xor   ecx, ecx
  77f237e9  call  eax // _Kernel32ThreadInitThunkFunction
  77f237eb  mov   [ebp+ms_exc.registration.TryLevel], 0fffffffeh // Leaving the __try block
  77f237f2  call  __SEH_epilog4
  77f237f7  retn  8

  77ec5e77 loc_77ec5e77:
  77ec5e77  call  [ebp+8]
  77ec5e7a  push  eax
  77ec5e7b  call  _RtlExitUserThread@4
```

Who Installs the Fallback Handler? III

When we rewrite the assembly back into C, we obtain the following:

```
void __stdcall __RtlUserThreadStart(
  DWORD (__stdcall* pfnThreadProc)(LPVOID),
  LPVOID pThreadArg
)
{
  __try
  {
    DWORD dwResult;

    if( Kernel32ThreadInitThunkFunction )
      dwResult = Kernel32ThreadInitThunkFunction( NULL, pfnThreadProc, pThreadArg );
    else
    {
      dwResult = pfnThreadProc( pThreadArg );
      RtlExitUserThread(dwResult);
    }
  }
  __except( RtlpGetExceptionFilter( GetExceptionInformat() ) )
  {
    ZwTerminateProcess( GetCurrentProcess(), GetExceptionCode() );
  }
}
```

Structured Exception Handling — Exploits I

SEH-based exploits overwrite the `EXCEPTION_REGISTRATION`.`handler` field and cause an exception. The topmost handler from `EXCEPTION_REGISTRATION`, pointed to by `FS:[0]`, which is usually the same as the one in the current function that got overwritten, gets called to decide whether it wishes to handle the exception caused by the attacker or not. The `handler` field is overwritten with an address of a pop-pop-`ret` instruction sequence:

```
pop whichever_register  // Remove the return address
pop whichever_register  // Remove the first parameter i.e. ExceptionRegistration
ret                     // Return to the address pointed to by EstablisherFrame
```

Why pop-pop-ret and **where** does the code return?

Structured Exception Handling — Exploits II

The call stack at the handler entry is as follows: (Windows 7)

```
application.exe!_except_handler
  ntdll.dll!ExecuteHandler2
    ntdll.dll!ExecuteHandler
      ntdll.dll!_RtlDispatchException
        ntdll.dll!_KiUserExceptionDispatcher
          application.exe!FAULTING_FUNCTION
```

```
EXCEPTION_DISPOSITION __cdecl except_handler(
  struct _EXCEPTION_RECORD* ExceptionRecord,
  void* EstablisherFrame,
  struct _CONTEXT* ContextRecord,
  void* DispatcherContext
);
```

The stack at the entry into `_except_handler` looks like this:

```
           Return             Ex...ionRecord*  Est...erFrame*   Cont...Record*
           address to                          i.e. ptr. to
           ExecuteHandler2                     EXC_REGISTR.*
0x0012F998 f9 71 92 77        80 fa 12 00      c8 fe 12 00      9c fa 12 00
0x0012F9A8 54 fa 12 00        c8 fe 12 00      0d 72 92 77      c8 fe 12 00
           Disp...Context*
```

The pop-pop-ret instructions (opcodes `5x 5y c3`, $x, y \in \{8, \ldots, f\} \setminus \{c\}$[2]) first remove the two top items off the stack (the return address and `ExceptionRecord*`) and the `ret` instruction jumps at the start of the `EXCEPTION_REGISTRATION`, where the `prev` field is. This field has been filled with the first 4 bytes of the exploit code.

Note: This approach requires an executable stack.

[2] `5c` is `pop esp` and is not appropriate for this scenario.

Epilogue

Epilogue is the standard ending of a function. Its purpose is to destroy the stack frame, restore non-volatile registers, and return to the caller, optionally cleaning-up parameters from the stack if the calling convention requires this.

Additionally, if stack-canaries are used, then the canary is verified and if there's a mismatch the executable is terminated.

A typical epilogue (MSVC/IA-32) with `__cdecl` convention

```
mov %ebp, %esp
pop %ebp
ret
```

A typical epilogue (MSVC/IA-32) with `__stdcall` convention

```
mov %ebp, %esp
pop %ebp
ret $0x8
```

A typical epilogue (GNU/IA-32)

```
leave
ret
```

A typical epilogue (GNU/x86_64)

```
leaveq
retq
```

Summary

Now, when we look at a function, we should be able to quickly tell:

- prologue
  - where does it start?
  - how many bytes of local variables are there?
  - is the stack pointer aligned?
  - does the function use the frame pointer?
  - does the function use a canary?
  - does the function use SEH?
  - if the function expects a parameter in `ECX`, it is likely to be `__thiscall` (a method) or `__fastcall` (and may expect another parameter in `EDX`).
- the body, starting right after the prologue, will be analyzed later! :-)
- epilogue
  - where does the function end?
  - does the function check the canary?
  - does the function restore the `EXCEPTION_REGISTRATION`?
  - if the function uses `ret XXX` it is likely to use `__stdcall` or `__thiscall`, otherwise it's likely to use `__cdecl` convention.

Initial Stack Frame Analysis I

A CFG [Lec 2] will give us a basic idea about the code logic. Now we try to better understand the stack frame by analyzing all instructions that could possibly interact with it (e.g. `mov`, `push`, `lea`, ...). This analysis partitions the section allocated by the `sub esp, __LOCAL_SIZE`[3] instruction. Each of the inspected instructions is checked for:

1. an offset — a position of the item in the stack frame;
2. a size modifier (byte, word, dword, ...) — gives the size of the item.

The stack frame is partitioned only to items of a **yet unknown data type and signedness** i.e. we should use only `UINT64`, `DWORD`, `WORD`, and `BYTE` data types. Since we do not know names of the local variables, we usually assign them a generic name such as `local_offset` for local variables and `arg_offset` for parameters. In the subsequent analyses, we will use `l_off` and `a_off` to conserve space.

[3] MSVC provides a compiler symbol `__LOCAL_SIZE` — a number of bytes occupied by local variables in the current function. This is useful when creating a function including its prologue and epilogue with `__declspec(naked)`.

Initial Stack Frame Analysis II

Let's partition a sample stack frame:

```
00412ad0  push  ebp
00412ad1  mov   ebp,esp
00412ad3  sub   esp,25Ch                  // Allocate 25C bytes for locals
00412ad9  mov   eax,dword ptr [00419000]  // Load __security_cookie in EAX
00412ade  xor   eax,ebp
00412ae0  mov   dword ptr [ebp-4],eax     // Store the canary
00412ae3  push  ebx
00412ae4  push  esi
00412ae5  push  edi                       // End of prologue
```

After executing the prologue the stack frame would look like this:

```
+-----+-----+-----+----...----+-----+-----+-----+--------------------------...
| edi | esi | ebx | local vars | can | ebp | ret | args and superior frames
+-----+-----+-----+----...----+-----+-----+-----+--------------------------...
↑                 ↑           ↑     ↑           ↑
esp               ebp-25C     ebp-4 ebp         ebp+8
```

where can is the canary and ret is the return address.

Initial Stack Frame Analysis III

Let's continue with the next chunk:

```
00412ae6  mov   dword ptr [ebp-8],0
00412aed  mov   dword ptr [ebp-0Ch],0
00412af4  mov   dword ptr [ebp-10h],0
00412afb  mov   dword ptr [ebp-21Ch],0
00412b05  lea   eax,[ebp-218h]
```

The first instruction stores a DWORD zero at [ebp-8]. This place is then a DWORD. The same repeats for [ebp-C], [ebp-10], and [ebp-21C].

```
...--+----...----+----+----...----+----+----+----+----+----+---...
     |           |l_21c| buf_218   |l_010|l_00c|l_008| can | ebp |
...--+----...----+----+----...----+----+----+----+----+----+---...
     ↑           ↑    ↑           ↑    ↑    ↑    ↑    ↑
     ebp-25c     |    ebp-218     |    ebp-c|    ebp-4|
                 ebp-21c          ebp-10    ebp-8     ebp
```

There are 4 local variables `l_21c`, `l_010`, `l_00c`, and `l_008`, each DWORD. The `lea` instruction calculates an address of a buffer of an unknown length starting at `ebp-218`, which we denote as `buf_218`.

Initial Stack Frame Analysis IV

Let's continue with the next chunk:

```
00412b0b  push  eax
00412b0c  push  0
00412b0e  push  0
00412b10  push  25h
00412b12  push  0
00412b14  call  dword ptr [Tokens!_imp__SHGetFolderPathW (0041a17c)]
00412b1a  mov   dword ptr [ebp-21Ch],eax
00412b20  cmp   dword ptr [ebp-21Ch],0
00412b27  jge   loc_00412b2d
00412b29  jmp   loc_00412b97
```

We can observe that the program calls the `SHGetFolderPathW` API. This is a public Win32 API from `shell32.dll`. We could derive both the buffer size and type immediately from it, as we know from the previous instruction that the register `EAX` points to the start of `buf_218`.

Initial Stack Frame Analysis V

Let's continue with the next chunk:

```
00412b0b  push  eax
00412b0c  push  0
00412b0e  push  0
00412b10  push  25h
00412b12  push  0
00412b14  call  dword ptr [Tokens!_imp__
00412b1a  mov   dword ptr [ebp-21Ch],eax
00412b20  cmp   dword ptr [ebp-21Ch],0
00412b27  jge   loc_00412b2d
00412b29  jmp   loc_00412b97
```

```
SHFOLDERAPI SHGetFolderPathW(
  HWND hwnd,
  int csidl,
  HANDLE hToken,
  DWORD dwFlags,
  LPWSTR pszPath
);
// pszPath MAX_PATH=260×2 B long
```

We can observe that the program calls the `SHGetFolderPathW` API. This is a public Win32 API from `shell32.dll`. We could derive both the buffer size and type immediately from it, as we know from the previous instruction that the register `EAX` points to the start of `buf_218`. The function has the following prototype, but we will not use this information for now and return to API calls later.

Initial Stack Frame Analysis VI

```
00412b2d  push   offset Tokens!'string' (00417a18)
00412b32  lea    eax,[ebp-218h]
00412b38  push   eax
00412b39  call   dword ptr [Tokens!_imp__PathAppendW (0041a1ac)]
00412b3f  lea    eax,[ebp-218h]
00412b45  push   eax
00412b46  call   dword ptr [Tokens!_imp__LoadLibraryW (0041a054)]
00412b4c  mov    dword ptr [ebp-8],eax
00412b4f  cmp    dword ptr [ebp-8],0
00412b53  jne    loc_00412b64
```

We can see a call to `PathAppendW` with our buffer and then a call to `LoadLibraryW` with our buffer. The return value of the `LoadLibraryW` API in register `EAX` is stored to `l_008`. The result is compared to zero and here the code block ends.

Initial Stack Frame Analysis VII

```
00412b55  mov   dword ptr [ebp-10h],0
00412b5c  jmp   loc_00412b97
```

Here is the **else** branch of the **if** statement found at the end of the previous code block that sets `l_010` to 0. This result is also compared to zero.

```
00412b64  push  offset Tokens!'string' (00417a38)
00412b69  mov   eax,dword ptr [ebp-8]
00412b6c  push  eax
00412b6d  call  dword ptr [Tokens!_imp__GetProcAddress (0041a044)]
00412b73  mov   dword ptr [ebp-0Ch],eax
00412b76  cmp   dword ptr [ebp-0Ch],0
00412b7a  je    loc_00412b86
```

This code block calls `GetProcAddress` and stores the result in `l_00c`.

Initial Stack Frame Analysis VIII

```
00412b7c  mov   eax,dword ptr [ebp+8]
00412b7f  push  eax
00412b80  call  dword ptr [ebp-0Ch]
00412b83  mov   dword ptr [ebp-10h],eax
00412b86  mov   eax,dword ptr [ebp-8]
00412b89  push  eax
00412b8a  call  dword ptr [Tokens!_imp__FreeLibrary (0041a048)]
00412b90  mov   dword ptr [ebp-8],0
```

This code block accesses `ebp+8`, where the first parameter is located and it is a `DWORD`. Next a function whose address is stored in `l_00c` is called. Ultimately, `FreeLibrary` is called, and variable `l_008` is zeroed.

```
...--+-----+----...----+-----+-----+-----+-----+-----+-----+-----+---...
     |l_21c| buf_218   |l_010|l_00c|l_008| can | ebp | ret |a_008|
...--+-----+----...----+-----+-----+-----+-----+-----+-----+-----+---...
     ↑     ↑           ↑     ↑     ↑     ↑     ↑     ↑     ↑
     |     ebp-218     |     ebp-c |     ebp-4 |     ebp+4 |
  ebp-21c           ebp-10      ebp-8         ebp         ebp+8
```

Initial Stack Frame Analysis IX

The last portion of the function's body sets the EAX register to the value of `l_010`. This is `return l_010;`.

```
00412b97  mov   eax,dword ptr [ebp-10h]
```

The final portion is the epilogue, which restores the registers, checks the canary, and returns. The function has at least one parameter `a_008`. There're no references to implied parameters in either ECX or EDX. The function also uses a `ret` instruction without an immediate operand. From this we infer the `__cdecl` calling convention.

```
00412b9a  pop   edi
00412b9b  pop   esi
00412b9c  pop   ebx
00412b9d  mov   ecx,dword ptr [ebp-4]
00412ba0  xor   ecx,ebp
00412ba2  call  Tokens!ILT+30(__security_check_cookie (00411023)
00412ba7  mov   esp,ebp
00412ba9  pop   ebp
00412baa  ret
```

Initial Stack Frame Analysis X

The final stack frame is:

```
+-----+-----+-----+-----...-----+-----+-----...-----+-----+-----+-----+-----+-----+-----+-----+-------...
| ebx | esi | edi | unused      |l_21c| buf_218     |l_010|l_00c|l_008| can | ebp | ret |a_008| prev. fms
+-----+-----+-----+-----...-----+-----+-----...-----+-----+-----+-----+-----+-----+-----+-----+-------...
↑                               ↑     ↑             ↑     ↑     ↑     ↑     ↑     ↑     ↑     ↑
esp                             |     ebp-218       |     ebp-c |     ebp-4 |     ebp+4 |     ebp+c
                                ebp-21c             ebp-10      ebp-8       ebp         ebp+8
```

So far we have:

```
DWORD __cdecl UnknownFunction( DWORD a_008 )
{
  DWORD  l_21c;            // ebp-21c
  BYTE   l_218[unknown];   // ebp-218
  DWORD  l_010;            // ebp-010
  DWORD  l_00c;            // ebp-00c
  DWORD  l_008;            // ebp-008

  ...

  return l_010;
}
```

Signedness Analysis I

The information we have so far extracted does not tell us much about data types other than their length. Now we should analyze all data types for their signedness. The easiest way is to examine all arithmetic instructions and their corresponding `cmp` and `jxx` instructions. If we see a `cmp/test` instruction or an arithmetic operation followed by a conditional jump instruction based on the **CF** flag, then we know the data type is unsigned, if based on **SF** and/or **OF** flag, then the data type is signed. Let's look at our code:

```
00412b20  cmp   dword ptr [ebp-21Ch],0
00412b27  jge   loc_00412b2d
00412b29  jmp   loc_00412b97
```

Since `jge` jumps if **SF**=**OF**, the data type of `l_21c` is signed and we can change `DWORD` to `int`. Unfortunately there are no other such `jxx` instruction so we cannot tell more at this point.

Signedness Analysis II

Instruction	CF	ZF	SF	OF	Mnemonic	Signed/Unsigned
`ja/jnbe`	0	0			Jump if above	unsigned
`jae/jnc`	0				Jump if above or equal	unsigned
`jb/jc/jnae`	1				Jump if below	unsigned
`jbe/jna`	$CF \lor ZF$				Jump if below or equal	unsigned
`je/jz`		1			Jump if equal	no information
`jne/jnz`		0			Jump if not equal	no information
`jg/jnle`		0	$SF = OF$		Jump if greater	signed
`jge/jnl`			$SF = OF$		Jump if greater or equal	signed
`jl/jnge`			$SF \neq OF$		Jump if less	signed
`jle/jng`		$ZF \lor (SF \neq OF)$			Jump if less or equal	signed
`jo`				1	Jump if overflow	signed
`jno`				0	Jump if not overflow	signed
`js`			1		Jump if sign	signed
`jns`			0		Jump if not sign	signed

Table: Conditional jump instructions and their use in determination of type signedness.

API Call Analysis I

The function we are analyzing calls external modules. These calls are to functions from the documented Windows API. We can infer parameter types, lengths and **content** from the API calls. This allows us to rectify the buffer `buf_218` and identify data types of all yet unknown data types.

1. If we return to slide no. 57, we can see that the last parameter of the function is a buffer exactly `MAX_PATH` characters long. Since `MAX_PATH` is 260 characters and each character is 2 bytes long, the buffer is 520 ($=(208)_{16}$) bytes long and occupies the entire `buf_218` field. Its declaration is then `WCHAR buf_218[MAX_PATH]`.
2. Next, the result of `SHGetFolderPathW` is of the `HRESULT` data type. This data type is signed (we already know that) and is exactly 4 bytes long (we also know that). We can change the `int` to `HRESULT`.
3. A call to `PathAppendW` concatenates a string to a path with the result totalling up to `MAX_PATH` characters — nothing new.

API Call Analysis II

4. A call to `LoadLibraryW`. This API takes a path of a PE module to load, loads it and returns its `HMODULE`. The result is stored in `l_008` so we can change its data type to `HMODULE`.
5. A call to `GetProcAddress`. This API takes an `HMODULE` as the first argument and `char*`, a symbol name to load, as the second argument. The function returns a pointer to the symbol. This pointer is stored in `l_00c` and we know it is a pointer to a function. The parameter passed to this function tells us the API name (`SetProcessDEPPolicy`) and further rectifies the `l_00c`'s type by telling us the calling convention (`__stdcall`) and the return type `BOOL`.
6. The function pointer stored in `l_00c` is called and one argument `a_008` is passed to it. The function returns a `DWORD` (but we know it is a 4-byte `BOOL`), which is stored in `l_010`.

API Call Analysis III

```
DWORD __cdecl UnknownFunction( DWORD a_008 )
{
  HRESULT  l_21c;                              // ebp-21c, item 2
  WCHAR    l_218[MAX_PATH];                    // ebp-218, item 1
  DWORD    l_010;                              // ebp-010, item 6
  BOOL     (__stdcall *l_00c)(DWORD);          // ebp-00c, item 5
  HMODULE  l_008;                              // ebp-008, item 4

  ...

  return l_010;
}
```

```
+----+----+----+----...----+----+----...----+----+----+----+----+----+----+----+-------...
| ebx | esi | edi | unused    |l_21c| buf_218    |l_010|l_00c|l_008| can | ebp | ret |a_008| prev. fms
+----+----+----+----...----+----+----...----+----+----+----+----+----+----+----+-------...
↑                          ↑    ↑               ↑    ↑    ↑    ↑    ↑    ↑    ↑    ↑
esp                        |    ebp-218         |    ebp-c|    ebp-4|    ebp+4|    ebp+c
                           ebp-21c              ebp-10    ebp-8     ebp       ebp+8
```

API Call Analysis IV

After renaming the labels and applying the initialization of the variables to zero, we get the following:

```
DWORD __cdecl UnknownFunction( DWORD a_008 )
{
  HRESULT  hr = S_OK;                        // ebp-21c, item 2
  WCHAR    wszLibraryPath[MAX_PATH];         // ebp-218, item 1
  DWORD    dwResult = 0;                     // ebp-010, item 6
  // ebp-00c, item 5
  BOOL     (__stdcall *pfnSetProcessDEPPolicy)(DWORD) = NULL;
  HMODULE  hLibrary = NULL;                  // ebp-008, item 4

  ...

  return dwResult;
}
```

Bibliography I

- Wikipedia Foundation, Inc.: *Software Engineering*, 2015, https://en.wikipedia.org/wiki/Software_engineering.
- Chikofsky, E. J. and Cross, J. H.: *Reverse engineering and design recovery: A taxonomy*, IEEE Software 7: 13-17, 1990, http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf
- Fog A.: *Calling conventions for different C++ compilers and operating systems*, 2014, http://www.agner.org/optimize/calling_conventions.pdf.
- Pietrek M.: *A Crash Course on the Depths of Win32™ Structured Exception Handling*, 1997, http://www.microsoft.com/msj/0197/Exception/Exception.aspx.
- Russinovich M., Solomon D. A., Ionescu A.: *Windows Internals Part 1*, 6^{th} ed., 2012.

Bibliography II

Skochinsky, I.: *Compiler Internals: Exceptions and RTTI*, Recon, 2012, https://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf