Software ‘Security’ {

[Function Hooking]

< Presented by
Augustin Rousset-Rouviere / Matteo Daluz /
Yohann Martin / Rohail Shabbir / Nathan Longa
>

}

introduction.md

summary.md

Table Of 'Contents' {

}

basic-asm-title.md

basic-asm-1.md

01 {

[Basic introduction to some Assembly concepts]

< All you need to understand
function hooking internal concepts >

}

Intel Assembly x86 < /1 > {

Operands order:
opcode destination, source

Some instructions to know:
JMP <relative address>

NOP

PUSH <register | address>

POP <register | address>

MOV <destination>, <source>

}

Register	Purpose
EAX	Accumulator register
ECX	Counter register
EDX	Data register
EBX	Base register
ESP	Stack pointer register
EBP	Stack base pointer register
ESI	Source index register
EDI	Destination index register

Intel Assembly x86 < /2 > {

Assembly instructions are bytes array:

JMP 0×40000 ⇒ 0×E9 0×00 0×00 0×00 0×40

NOP ⇒ 0×90

Each instruction have an RAM address (can be dynamic or static):

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08]
0×0406	03 C8	add ecx, eax

}

What is a function? < /1 > {

```
int sum(int a, int b) {
    return a + b;
}
```

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08] // a
0×0406	03 45 0C	add eax, [ebp+0C] // b
0×0409	5D	pop ebp
0×040A	C3	ret

}

02 {

[The Function Hooking and Trampoline]

< The art of detouring a function >

}

What is a detour? < /2 > {

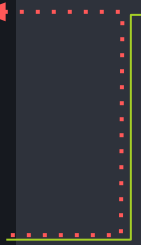
Program target:

```
int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int res = sum(1, 2);
    return 0;
}
```

Malicious attacker code:

```
int fake_sum(int a, int b) {
    return a - b;
}
```



}

How to detour? < /3 > {

`sum(int, int)` function:

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08] // a
0×0406	03 45 0C	add eax, [ebp+0C] // b
0×0409	5D	pop ebp
0×040A	C3	ret

`fake_sum(int, int)` function:

Address	Bytes	Opcode
0×0800	55	push ebp
0×0801	8B EC	mov ebp, esp
0×0803	8B 45 08	mov eax, [ebp+08] // a
0×0806	2B 45 0C	sub eax, [ebp+0C] // b
0×0809	5D	pop ebp
0×080A	C3	ret

Remember the JMP instruction: JMP <relative address>

}

How to detour? < /3 > {

`sum(int, int)` function:

Address	Bytes	Opcode
0×0400	55	jmp fake_sum
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08] // a
0×0406	03 45 0C	add eax, [ebp+0C] // b
0×0409	5D	pop ebp
0×040A	C3	ret

`fake_sum(int, int)` function:

Address	Bytes	Opcode
0×0800	55	push ebp
0×0801	8B EC	mov ebp, esp
0×0803	8B 45 08	mov eax, [ebp+08] // a
0×0806	2B 45 0C	sub eax, [ebp+0C] // b
0×0809	5D	pop ebp
0×080A	C3	ret

But the JMP instruction take 5 bytes:
JMP 0×40000 ⇒ 0×E9 0×00 0×00 0×00 0×40

}

How to detour? < /3 > {

`sum(int, int)` function:

Address	Bytes	Opcode
0×0400	E9 FB 03 00 00	jmp fake_sum
0×0405	08	? ?
0×0406	03 45 0C	add eax, ...
0×0409	5D	pop ebp
0×040A	C3	ret

`fake_sum(int, int)` function:

Address	Bytes	Opcode
0×0800	55	push ebp
0×0801	8B EC	mov ebp, esp
0×0803	8B 45 08	mov eax, [ebp+08] // a
0×0806	2B 45 0C	sub eax, [ebp+0C] // b
0×0809	5D	pop ebp
0×080A	C3	ret

The NOP instruction to the rescue!

}

How to detour? < /3 > {

`sum(int, int)` function:

Address	Bytes	Opcode
0×0400	E9 FB 03 00 00	jmp fake_sum
0×0405	90	nop
0×0406	03 45 0C	add eax, ...
0×0409	5D	pop ebp
0×040A	C3	ret

`fake_sum(int, int)` function:

Address	Bytes	Opcode
0×0800	55	push ebp
0×0801	8B EC	mov ebp, esp
0×0803	8B 45 08	mov eax, [ebp+08] // a
0×0806	2B 45 0C	sub eax, [ebp+0C] // b
0×0809	5D	pop ebp
0×080A	C3	ret

Our hook is a success!

}

The detour method code < /4 > {

```
bool DetourFunction(void * src, void * dst, int len)
{
    if (len < 5) return false;

    memset(src, 0×90, len);

    uintptr_t relativeAddress = ((uintptr_t)dst - (uintptr_t)src) - 5;

    *(BYTE*)src = 0×E9;
    *(uintptr_t*)((uintptr_t)src + 1) = relativeAddress;

    return true;
}
```

}

But can we go further? < /5 > {

Main drawback: No longer access to the original function

Consequence: We can not make any pre/post patching

Pre-patching (args):

```
int fake_function(int a) {
    a = a + 1;

    return original_func(a);
}
```

Post-patching (returned value):

```
int fake_function(int a) {
    int res = original_func(a);

    return res - 1;
}
```

}

The trampoline solution < /6 > {

The solution: The trampoline!

}

What is a trampoline? < /7 > {

sum(int, int) function:

Address	Bytes	Opcode
0×0400	E9 FB 03 00 00	jmp fake_sum
0×0405	90	nop
0×0406	03 45 0C	add eax, ...
0×0409	5D	pop ebp
0×040A	C3	ret

trampoline(int, int) function:

Address	Bytes	Opcode
0×0900	55	push ebp
0×0901	8B EC	mov ebp, esp
0×0903	8B 45 08	mov eax, ...
0×0906	E9 00 00 00 00	jmp sum+6

}

What is a trampoline? < /7 > {

sum(int, int) function:

Address	Bytes	Opcode
0×0400	E9 FB 03 00 00	jmp fake_sum
0×0405	90	nop
0×0406	03 45 0C	add eax, ...
0×0409	5D	pop ebp
0×040A	C3	ret

trampoline(int, int) function:

Address	Bytes	Opcode
0×0900	55	push ebp
0×0901	8B EC	mov ebp, esp
0×0903	8B 45 08	mov eax, [ebp+08]
0×0909	E9 00 00 00	jump sum+6

}

fake_sum(int, int) function:

Address	Bytes	Opcode
0×0800	55	push ebp
0×0801	8B EC	mov ebp, esp
0×0803	8B 45 08	mov eax, [ebp+08]
0×0806	2B 45 0C	sub eax, [ebp+0C]
0×0809	E8 00 00 00	call trampoline
0×080D	5D	pop ebp
0×080E	C3	ret

The trampoline method code < /8 > {

```
char* TrampolineHook(char* src, char* dst, const intptr_t len)
{
	if (len < 5) return 0;

	void* gateway = VirtualAlloc(0, len + 5, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

	memcpy(gateway, src, len);

	intptr_t  gatewayRelativeAddr = ((intptr_t)src - (intptr_t)gateway) - 5;
	*(char*)((intptr_t)gateway + len) = 0xE9;
	*(intptr_t*)((intptr_t)gateway + len + 1) = gatewayRelativeAddr;

	DetourFunction(src, dst, len);

	return (char*)gateway;
}
```

}

What the use of function hooking?

< /9 > {

A technique mostly used in game hacking:

- Can be used for man-in-the-middle (MITM) attacks
  - Listen to the send() and recv() functions

- Can be used for detouring graphic engines

- Can be used for patching any game function to change its behavior

}

Use case: send() and recv() MITM

< /10 > {

In windows, two low-level functions are provided for sending/receiving data over a TCP connection:

```
ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t recv(int s, void *buf, ssize_t len, int flags);
```

We can hook those two functions to intercept all information exchanged between the game/program client and the game/program server.

By patching we can even change the contents sended or received by the game/program

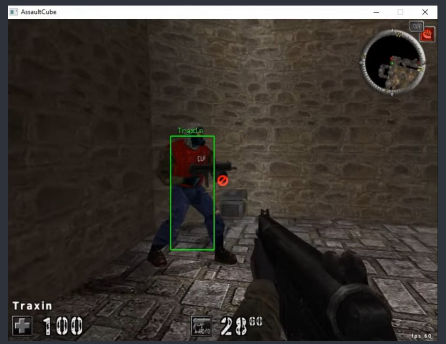
}

Use case: Graphic Engine

< /11 > {

For games, it is interesting to hook some engine functions to be able to add our own interface on top of the game for our malicious program.

For example with the engine OpenGL we can hook the function `wglSwapBuffers()` to add our own interface:



}

```
03 {
    [Live demonstration of trampoline
    hooking]
}
```

04 {

[Two methods to prevent function hooking]

< Just how to prevent from this nightmare ? >

}

protection-title.md

protection-summary.md

Obfuscation / Packer < /1 > {

< Let's make the assembly code impossible to understand! >

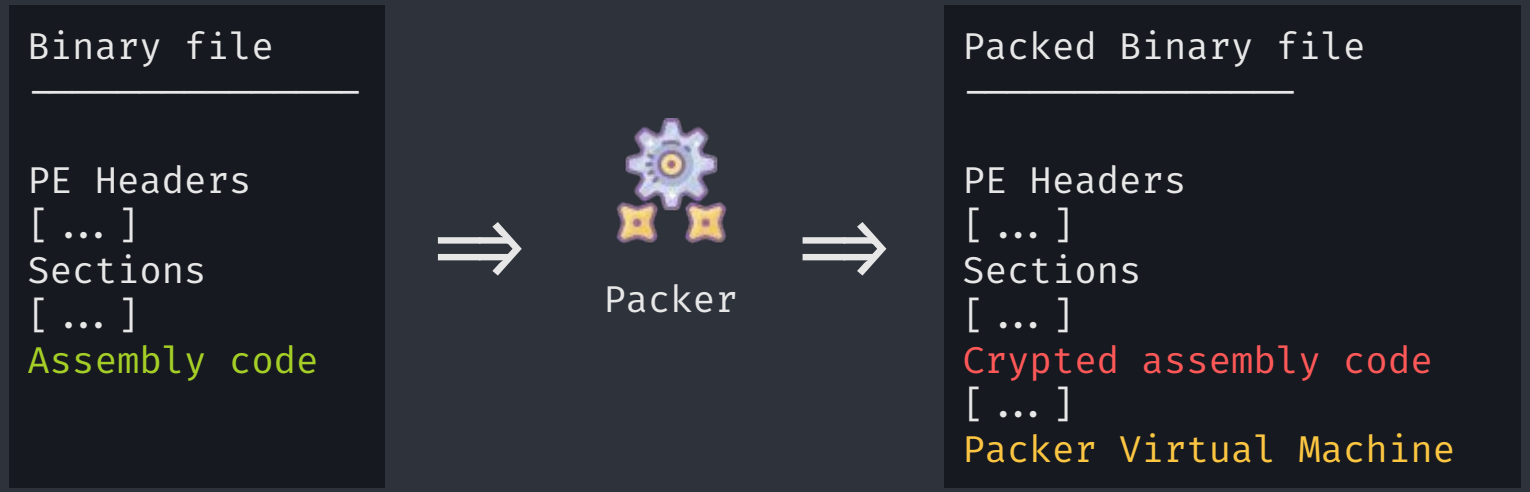
}

Memory fingerprint < /2 > {

< It is time to detect these unwanted changes! >

}

Obfuscation / Packer < /1 > {



}

Obfuscation / Packer < /1 > {

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, ebp
0×0406	03 C8	add ecx, eax

⇒

Address	Bytes	Opcode
0×0400	34	? ? ?
0×0401	22 01	? ? ?
0×0403	14 E5 B8	? ? ?
0×0406	75 F4	? ? ?

}

Memory fingerprint < /2 > {

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08]
0×0406	03 45 0C	add eax, [ebp+0C]
0×0409	5D	pop ebp
0×040A	C3	ret

Function payload: 55 8B EC 8B 45 08 03 45 0C 5D C3

}

Memory fingerprint < /2 > {

Address	Bytes	Opcode
0×0400	55	push ebp
0×0401	8B EC	mov ebp, esp
0×0403	8B 45 08	mov eax, [ebp+08]
0×0406	03 45 0C	add eax, [ebp+0C]
0×0409	5D	pop ebp
0×040A	C3	ret

SHA256(55 8B EC 8B 45 08 03 45 0C 5D C3) =

bd938e409fd7adea661f129903b0c4232179656738 32a58652967e992d90b850

}

Memory fingerprint < /2 > {

Address	Bytes	Opcode
0×0400	E9 E3 FF FF FF	jmp 00371000
0×0405	90	nop
0×0406	03 45 0C	add eax, [ebp+0C]
0×0409	5D	pop ebp
0×040A	C3	ret

SHA256(E9 E3 FF FF FF 90 03 45 0C 5D C3) =

e137dfc53100168f0aa460b2a9ae30db12c3ec49b577d41b5d4e2f44792c82fc

}

Memory fingerprint < /2 > {

SHA256(55 8B EC 8B 45 08 03 45 0C 5D C3)
≠
SHA256(E9 E3 FF FF FF 90 03 45 0C 5D C3)

bd938e409fd7adea661f129903b0c4232179656738 32a58652967e992d90b850
≠
e137dfc53100168f0aa460b2a9ae30db12c3ec49b577d41b5d4e2f44792c82fc

Function tampering detected!
Result: Program execution aborted

}

conclusion-questions.md

Thanks; {

‘Do you have any questions?’

References used:
https://guidedhacking.com/threads/how-to-hook-functions-code-detouring-guide.14185/

http://jbremer.org/x86-api-hooking-demystified/

https://is.muni.cz/th/qe1y3/bk.pdf

Course + Demo: https://github.com/Astropilot/FunctionHookingCourse

}

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

tochange.md

tochange.md

tochange.md

tochange.md