CSE 5474 — Software Security Spring 2022

BabyShell CTF

Due 11:59PM Feb 22nd, 2022

1 Overview

In this CTF assignment, you are given 14 challenges with varied difficulties (from `babyshell_level1` to `babyshell_level14`). Those challenges read in some bytes, modifies them (depending on the specific challenge configuration), and executes them as code, which is one of the most common exploitation scenarios, called "code injection". Through this series of challenges, you will practice your shellcode development skills under various constraints. Please use `pwndbg` instead of just `GDB`.

- `babyshell_level 1` (3 pts). In this challenge, you need to provide a shellcode from stdin. The software will directly execute the shellcode you provided. By executing the shellcode, you will get the flag. As an entry level challenge, it will print out walk-through information to guide you step by step. For example, the size of shellcode, the address where the shellcode will be placed are provided.

- `babyshell_level 2-4` (5 pts). In this challenge, you need to provide a shellcode from stdin. By executing the shellcode, the program will print out the flag. However, there is a constraint in shellcode. You need to provide shellcode satisfying these constraints (e.g., cannot use the specific char).

- `babyshell_level 5` (5 pts). In this challenge, you need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. You cannot use some instructions since it is the constraint of the software. However, it looks like that the memory is not protected. You might use this property to solve the challenge. **Hint**: Dynamically change the instruction like dyanmic code patching what we did on reverse level 6 and 7.

- `babyshell_level 6-7` (7 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. You cannot use some instructions since it is constraint of the software. Also, it looks like that the memory is protected. **Hint**: You can use NOP instruction. The NOP instruction does nothing but advances the EIP.

- `babyshell_level 8` (7 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. There is a constraint so that the content of the shellcode should satisfy the constraint.

- `babyshell_level 9` (9 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. It is similar to level 8, but one more constraint is added. You need to decompile and analyze the program to figure out what kinds of constraint exist in the program. **Hint**: You might use rax, ax, al, and so on. Add unused instructions to resolve the constraint.

- `babyshell_level 10` (9 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. The content of the shellcode should satisfy the constraint.
- `babyshell_level 11` (9 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. At the same time, the program closes stdin, stderr, and stdout before executing the shellcode. You need to print out the flag without stdin and stdout.
- `babyshell_level 12-13` (9 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. The size of the shellcode should satisfy the constraint. You should propose simple shellcode and print out the flag. **Hint**: Use chmod shellcode to change permission of flag file.
- `babyshell_level 14` (9 pts). You need to provide a shellcode from stdin. By executing the shellcode, you should get the flag. The size of the shellcode should satisfy the constraint. You should propose simple shellcode and print out the flag. **Hint**: Contrast to level 13, two stage shellcode injections are possible.

2 Tools and syscalls

This section will give you some options of tools and syscalls that can help you solve the CTF problems.

2.1 Pwntools

Overview. Pwntool (`https://github.com/Gallopsled/pwntools`) is a framework for solving CTF and developing exploits. For installation, you can quickly install it with in our virtual machine:

```
sudo apt-get update
sudo apt-get install python3 python3-pip python3-dev git
sudo apt-get install libssl-dev libffi-dev build-essential
sudo python3 -m pip install --upgrade pip
sudo python3 -m pip install --upgrade pwntools
```

If there are any problems, please post your question in piazza or refer to the issue (`https://github.com/Gallopsled/pwntools/issues`) of this repo.

Manual. Pwntools has a every good manual to help you learn it. You can find it here (`https://docs.pwntools.com/en/stable/`). In the manual, there are different modules, but we can solve the CTFs by using pwnlib.asm and pwnlib.shellcraft module. Note that you should choose the right pwnlib.shellcraft module based your CPU. You can find it out by one of the following two commands:

```
cat /proc/cpuinfo
uname -a
```

For instance, for the Intel x86-64 CPU, you can find the manual in pwnlib.shellcraft.amd64 (https://docs.pwntools.com/en/stable/shellcraft/amd64.html).

Tutorials. Pwntools can generate shellcode for you. For example, if want to open a file named 'flag', you can directly use the built-in method open(). The output of this function is the assembly codes as shown below:

```
>>> import pwnlib
>>> asm = pwnlib.shellcraft.amd64.open("flag")
>>> print(asm)
    /* open(file='flag', oflag=0, mode=0) */
    /* push b'flag\x00' */
    push 0x67616c66
    mov rdi, rsp
    xor edx, edx /* 0 */
    xor esi, esi /* 0 */
    /* call open() */
    push SYS_open /* 2 */
    pop rax
    syscall
```

Pwntools can also converts assembly codes to machine codes. For example:

```
>>> import pwnlib
>>> import pwn
>>> asm = pwnlib.shellcraft.amd64.open("flag")
>>> shellcode = pwn.asm(asm, arch='amd64', os='linux')
>>> print(shellcode)
b'hflagH\x89\xe71\xd21\xf6j\x02X\x0f\x05'
```

Moreover, you can directly use assembly code. For example:

```
>>> import pwn
>>> shellcode = pwn.asm(
    """
    xor edi, edi
    mov esi, edx
    """
    , arch='amd64', os='linux')
>>> print (shellcode)
b'1\xff\x89\xd6'
```

Pwntools is very useful for solving the first several CTF challenges. You can exploit and leverage it. Lastly, you can inject your input using pwntools as well. For example:

```
>>> from pwn import *
```

```
>>> import pwn
>>> asm = pwnlib.shellcraft.amd64.linux.open("flag")
>>> shellcode = pwn.asm(asm, arch='amd64', os='linux')
>>> p = process("./program")
>>> p.send(shellcode)
>>> p.interactive()
or
>>> f = open('shellcode', 'wb')
>>> f.write(shellcode)
>>> f.close()
./program < shellcode
```

Debug mode is following the below instead of just `process`.

```
>>> p = gdb.debug('./program',
    '''
    b main
    continue
    ''')
```

When you use `Python3` and execute `p.interactive()`, you may get the error `BrokenPipeError`. But, you can still get the flag.

2.2 Syscalls

Since you will write your own shell codes for reading the CTF flag, one of the most important steps is using syscalls. For example, if you want to open a file, you have to use open syscall. But there are many syscalls available for you, e.g., more than 300 syscalls for x86_64 (`https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit`). We provide below some further information about syscalls to help you solve the CTF challenges.

What is syscall? Syscall (in AMD64, X86-64) is a very basic assembly instruction for calling the system call whose assembly language interface has the specified number with the specified arguments. For example, the system call number that represents open file is 2 in x86_64. To make a system call in x86_64, you can set the system call number in register rax, and its arguments, in order, in rdi, rsi, rdx, r10, r8, and r9 registers, then invoke syscall. The following assembly codes are invoking open system call with syscall instruction:

```
/* open(file='flag', oflag=0, mode=0) */

push 0x67616c66 /* push b'flag\x00' */
mov rdi, rsp
xor edx, edx /* 0 */
xor esi, esi /* 0 */
/* call open() */
```

```
push SYS_open /* 2 */
pop rax
syscall
```

How to use a specific syscall? Since there are hundreds of system calls, we cannot remember each of them. Fortunately, the above section has provided some hints of available options for solving each CTF challenge. To understand the functionality, arguments and return values of a specific syscall, you can either search online or use man command. For example, you can find every detail of open syscall from this online page (`https://man7.org/linux/man-pages/man2/open.2.html`).

3 Deliverables

The flags you captured, and please submit them in `https://cse5474.osuseclab.com`, and the writeup describing how you solve these challenges.