OS Security III

Implicit Behavior - Code as Data

OS Security III

Theme:

How easy is it to reason about what your code does?

Implicit Behavior

Implicit Behavior

```
#define struct union
```

Implicit Behavior

Environment Changes Behavior of Code

Implicit Behavior

Environment Changes Behavior of Code

Reasoning about behavior is hard

Implicit Behavior

```
# marked setuid, owned by root

echo hi
```

Implicit Behavior

```
$ echo '#!/bin/bash' > echo
$ echo '/bin/bash' >> echo
$ export PATH=.:$PATH
$ ./setuid.sh
# whoami
root
```

Implicit Behavior

```
# marked setuid, owned by root

ls -l ~
```

Implicit Behavior

```
$ export HOME=/bob
$ ./setuid.sh
drwxr-xr-x ... juicy_secrets
drwxr-xr-x ... less_juicy_secrets
drwxr-xr-x ... secrets
```

Implicit Behavior: LD_PRELOAD

- `LD_PRELOAD` environment variable
- Linker treats as path to dynamic object

```
$ gcc -shared -fPIC mal.c -o mal
$ LD_PRELOAD=./mal sudo su
# whoami
root
```

Implicit Behavior: LD_PRELOAD

- Luckily, linkers now are better about this
- Wasn't always the case, though

Implicit Behavior: DYLD_PRINT...

- DYLD_PRINT_TO_FILE
- Vulnerability in OS X reported in 2015
- Environment variable tells loader what file to use for error logging (normally stderr)
- Linker doesn't check to see if it's running from a setuid/setgid binary
- Write to files as other users (e.g., root)

Implicit Behavior

- PATH
- HOME
- LD_PRELOAD
- DYLD_PRINT_TO_FILE

Code as Data

Code as Data

```
ls $USER
```

Code as Data

```
USER=-R
ls $USER
```

Code as Data

Code and Data Go in the Same Place

Code as Data

Code and Data Go in the Same Place

Reasoning about behavior is hard

Code as Data

- system()
- echo
- Shellshock

Code as Data: system()

```
int system(const char *cmd) {
   const char *args[4];
   args = {“/bin/sh”, “-c”, cmd, 0};
   execve(“/bin/sh”, args, NULL);
   ...
}
```

Code as Data: system()

```
system("echo hi");

/bin/sh -c echo hi
------- -- -------
argv[0] 1  2
```

Code as Data: system()

```
// marked setuid
int main(int argc, char *argv[]) {
  char *cmd =
    str_append("echo ", argv[1]);
  system(cmd);
}
```

Code as Data: system()

```
$ ./a.out ‘hello, world’
hello, world
```

Code as Data: system()

```
$ ./a.out ‘hello; cat /etc/shadow’
hello
root:!:16844:0:99999:7:::
daemon:*:16844:0:99999:7:::
bin:*:16844:0:99999:7:::
...
```

Code as Data: echo

```
for file in $(echo *); do
  echo "$file: $(cat $file)"
done
```

Code as Data: echo

```
# cwd has foo, bar, baz
for file in foo bar baz; do
  echo "$file: $(cat $file)"
done
```

Code as Data: echo

```
# cwd has -e, \x2Fetc\x2Fshadow
for file in $(echo *); do
  echo "$file: $(cat $file)"
done
```

Code as Data: echo

```
# cwd has -e, \x2Fetc\x2Fshadow
for file in
  $(echo -e, \x2Fetc\x2Fshadow); do
  echo "$file: $(cat $file)"
done
```

Code as Data: echo

```
# cwd has -e, \x2Fetc\x2Fpasswd
for file in /etc/shadow; do
  echo "$file: $(cat $file)"
done
```

Code as Data: Shellshock

- In bash, this is a simple function:

```
hi () {
   echo hi
}
```

- Or, more compactly:

```
hi () { echo hi; }
```

Code as Data: Shellshock

- `hi () { echo hi; }` is a statement
- Evaluating it adds `hi` to the environment (bash's environment, not the OS environment)
- At start, bash looks through OS environment for function-looking things
- Evalues them, adding them to its environment

Code as Data: Shellshock

- `hi () { echo hi; }` is a statement
- Evaluating it adds `hi` to the environment (bash's environment, not the OS environment)
- At start, bash looks through OS environment for function-looking things
- Evalues them, adding them to its environment
- So what's the problem?

Code as Data: Shellshock

- `hi='() { echo hi; }; echo bye'`

Code as Data: Shellshock

- `hi='() { echo hi; }; echo bye'`
- Steps:
  a. bash sees `hi` variable; looks like function
  b. evaluates it to add the function to the environment
  c. the semicolon ends the function definition
  d. begins evaluating new statement
  e. evaluates and runs `echo bye`