ISA564

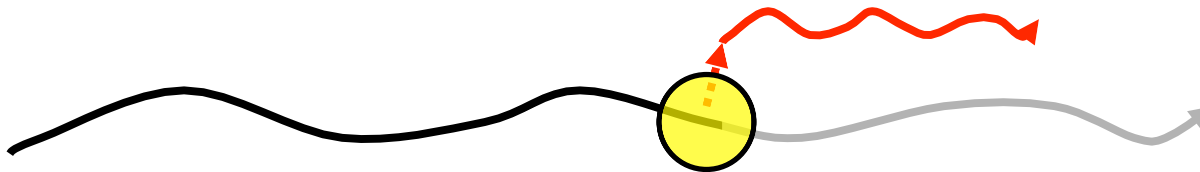
SECURITY LAB

Code Injection Attacks

- Anatomy of Code-Injection Attacks
- Lab 3: Buffer Overflow

Anatomy of Code-Injection Attacks

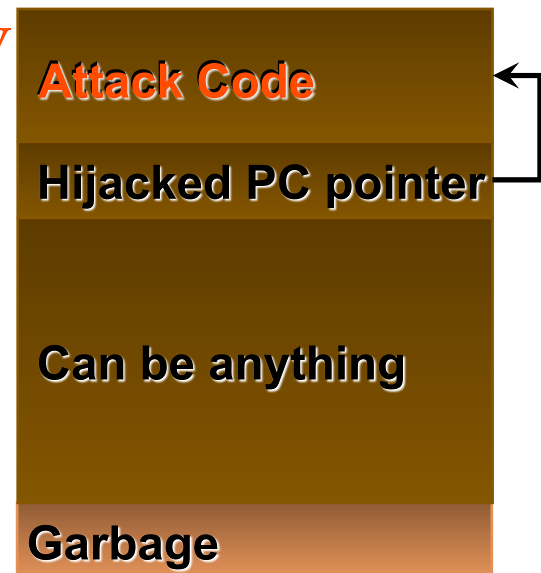
- Background
  - About 60% of CERT/CC advisories deal with unauthorized control information tampering [XKI03]



  - E.g.: Overflow buffer to overwrite return address
  - Other bugs can also divert control

Anatomy of Code-Injection Attacks

- Three essential stages
  1. Trigger software vulnerability
  2. Hijack control flow
  3. Execute attacker code



- We are not talking about social engineering attacks

Stage 1: Trigger Software Vulnerability

- What are those software vulnerabilities
  - Buffer overflows
  - Format-string vulnerability
  - Integer overflows
  - Double-free vulnerability
  - SQL injection vulnerability
  - XSS scripting vulnerability
  - …
- How to trigger them?
  - Different vulnerabilities usually require different ways to exploit them
  - We will focus on buffer overflows (Lab 3) and XSS/SQL Injection (Lab 6)

Stage 2: Hijack Control Flows

- How?
  - Overwriting control data
  - Overwriting non-control data
- What are those control data?
  - Return address
  - Function pointers
    - Windows: IAT/EAT, VTABLE
    - Linux: PLT/GOT

Stage 2: Hijack Control Flows (2)

- Control-data atttacks
  - Currently the most dominant form of memory corruption attacks [*CERT* and *Microsoft Security Bulletin*]

- Non-control-data attacks: attacks not corrupting any control data
  - i.e., attacks preserving the integrity of control flow of the victim process

Stack Overflow

- A buffer overflow occurs when you try to put too many bits into an allocated buffer.
- When this happens, the next contiguous chunk of memory is overwritten, such as
  - Return address
  - Function pointer
  - Previous frame pointer, etc.
- Also an attack code is injected.
- This can lead to a serious security problem.

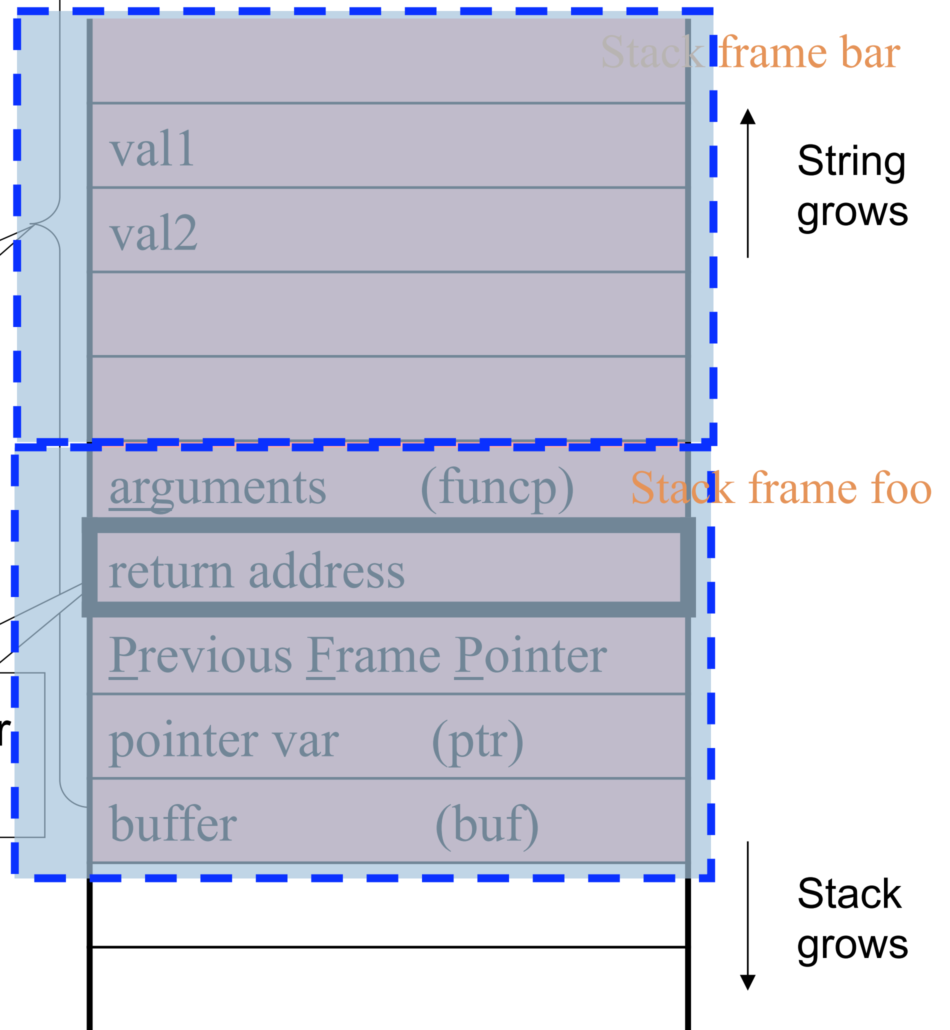
Stack Layout and Contaminated Memory by the Attack --- when function foo is called by bar.

```
int bar (int val1) {
    int  val2;
    foo (a_function_pointer);
}
```

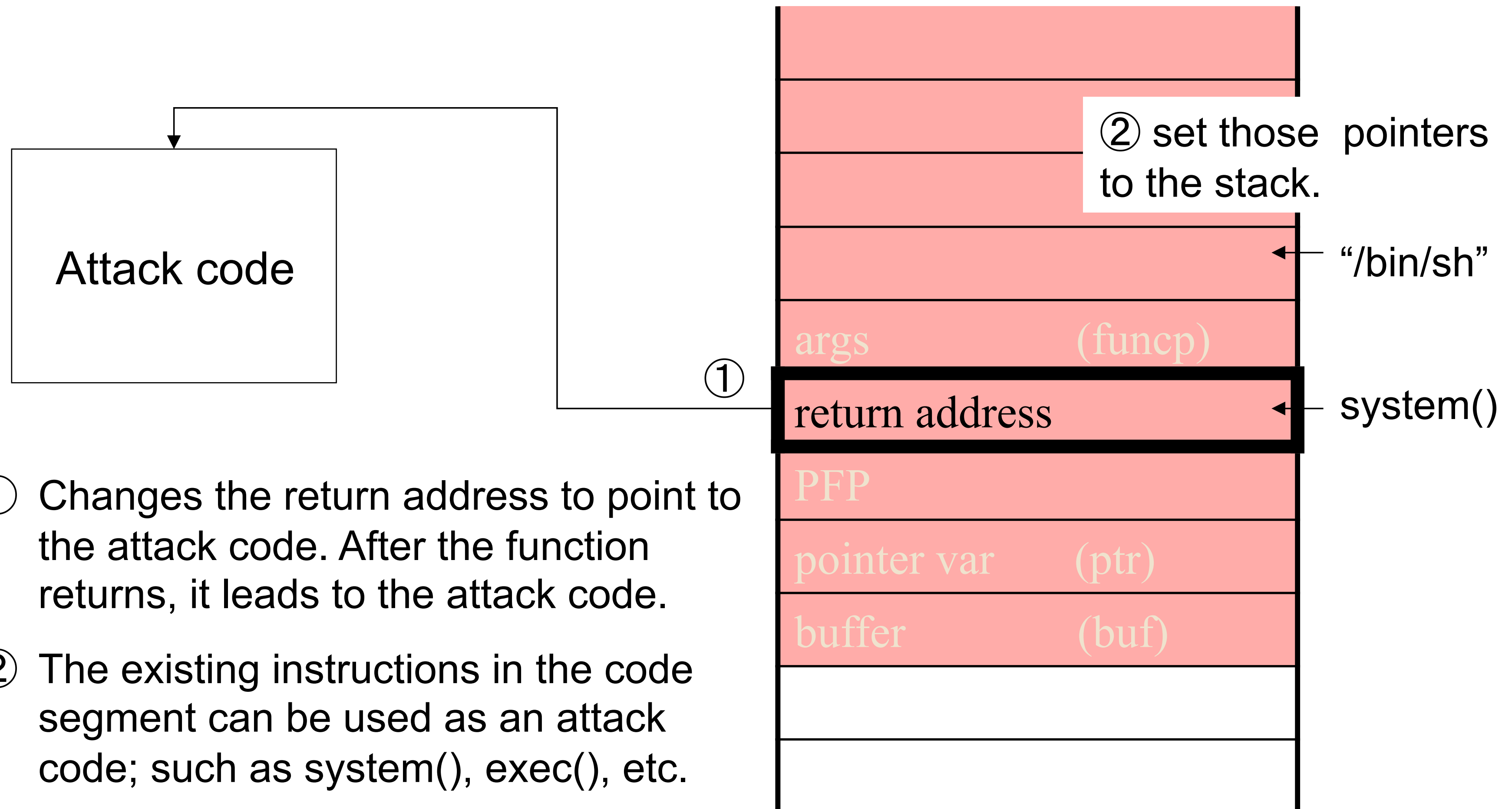
Contaminated memory

```
int foo (void (*funcp)()) {
    char* ptr = point_to_an_array;
    char buf[128];
    gets (buf);
    strncpy(ptr, buf, 8);
    (*funcp)();
}
```

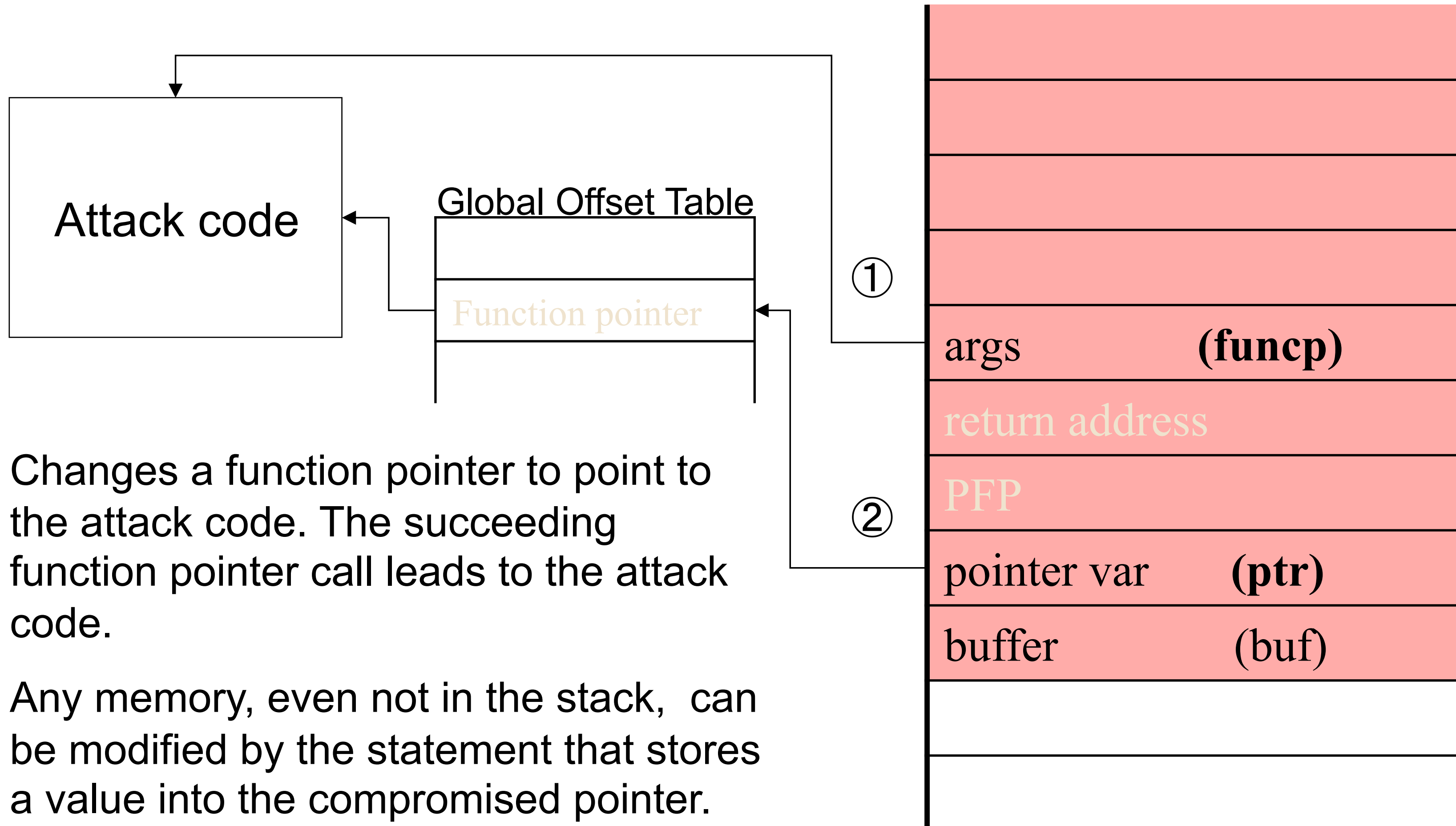
Most popular target



Attack Scenario #1
--- by changing the return address



Attack Scenario #2
--- by changing pointer variables

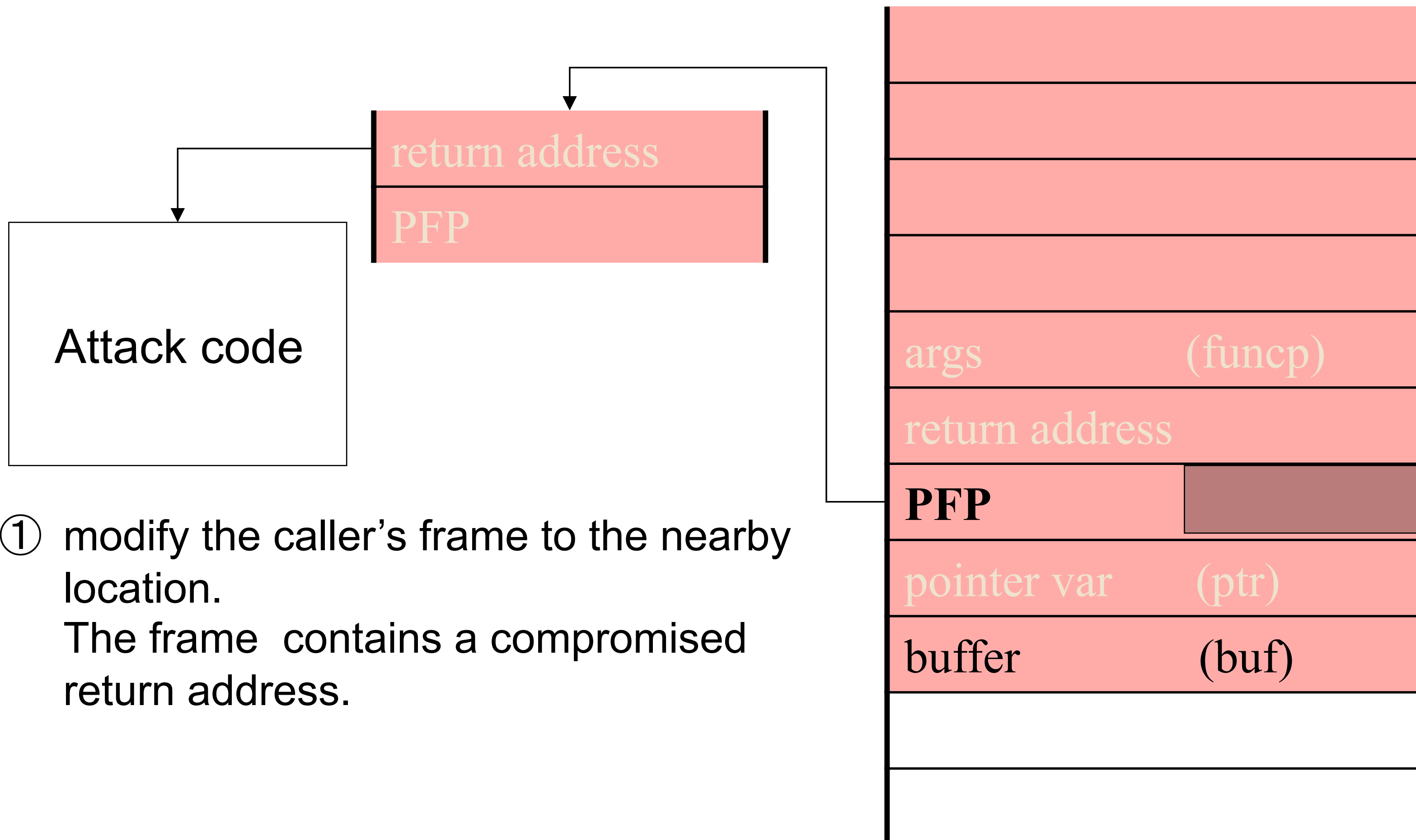


① Changes a function pointer to point to the attack code. The succeeding function pointer call leads to the attack code.

② Any memory, even not in the stack, can be modified by the statement that stores a value into the compromised pointer.

E.g.
```
strncpy(ptr, buf, 8);
*ptr = 0;
```

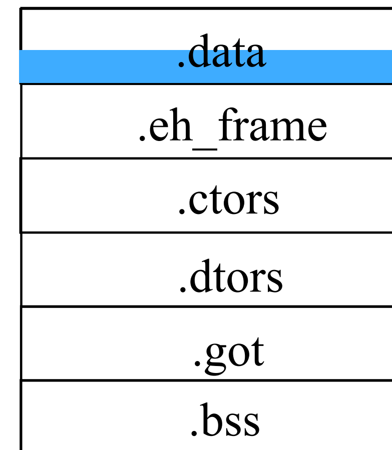
Attack Scenario #3
--- by changing the previous frame pointer

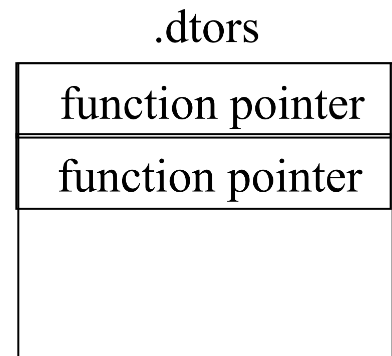


① modify the caller's frame to the nearby location.
The frame contains a compromised return address.

Data/BSS Overflow

- Data contains global or static compile-time initialized data
- Bss contains global or static uninitialized data
- Stored together with other sections that control program execution
- An attacker can overflow into other sections





- Heap overflows and double frees exist for many memory allocators
  - Dlmalloc (overflow, double free)
  - CSRI (overflow)
  - Quickfit (overflow)
  - Phkmalloc (overflow)
  - Boehm's Garbage Collector (overflow, double free)

Stage 2: Hijack Control Flows (2)

- Control-data atttacks
  - Currently the most dominant form of memory corruption attacks [*CERT* and *Microsoft Security Bulletin*]
- Non-control-data attacks: attacks not corrupting any control data
  - i.e., attacks preserving the integrity of control flow of the victim process

Example 1: Non-Control-Data Attack against *WU-FTPD* Server (via a format string bug)

```
int x;
FTP_service(...) {
  authenticate();
  x = user ID of the authenticated user;
  seteuid(x);
  while (1) {
    get_FTP_command(...);
    if (a data command?)
         getdatasock(...);
  }
}
```

x uninitialized, run as EUID 0

x=109, run as EUID 0

x=109, run as EUID 109. Lose the root privilege!

Get a special SITE EXEC command.
Exploit a format string vulnerability.
x= 0, still run as EUID 109.

When return to service loop, still runs as EUID 0 (root). Allow us to upload */etc/passwd*

We can grant ourselves the root privilege!

x=0, run as EUID 0

```
seteuid(x);
```

x=0, run as EUID 0

Only corrupt an integer, not a control data attack.

Example 2: Non-Control-Data Attack against *NULL-HTTP* Server (via a heap overflow bug)

- Attack the configuration string of CGI-BIN path.
- Mechanism of CGI
  - suppose server name = www.foo.com
    CGI-BIN = /usr/local/httpd/exe
  - Requested URL = http://www.foo.com/cgi-bin/bar
  - The server executes

The server gives me a root shell!
Only overwrite four characters in the CGI-BIN string.

- Our attack
  - Exploit the vulnerability to overwrite CGI-BIN to /bin
  - Request URL http://www.foo.com/cgi-bin/sh
  - The server executes

Example 3: Non-Control-Data Attack against *SSH Communications* SSH Server (via an integer overflow bug)

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

Stage 2: Hijack Control Flows (3)

- Discussion
  - Control-data attack ➔ unrelated to the semantics of the victim process
    - Hijack the control flow, do whatever you like
  - Non-control-data attack ➔ rely on the semantics of the victim process
    - Would this be a constraint for the attackers?

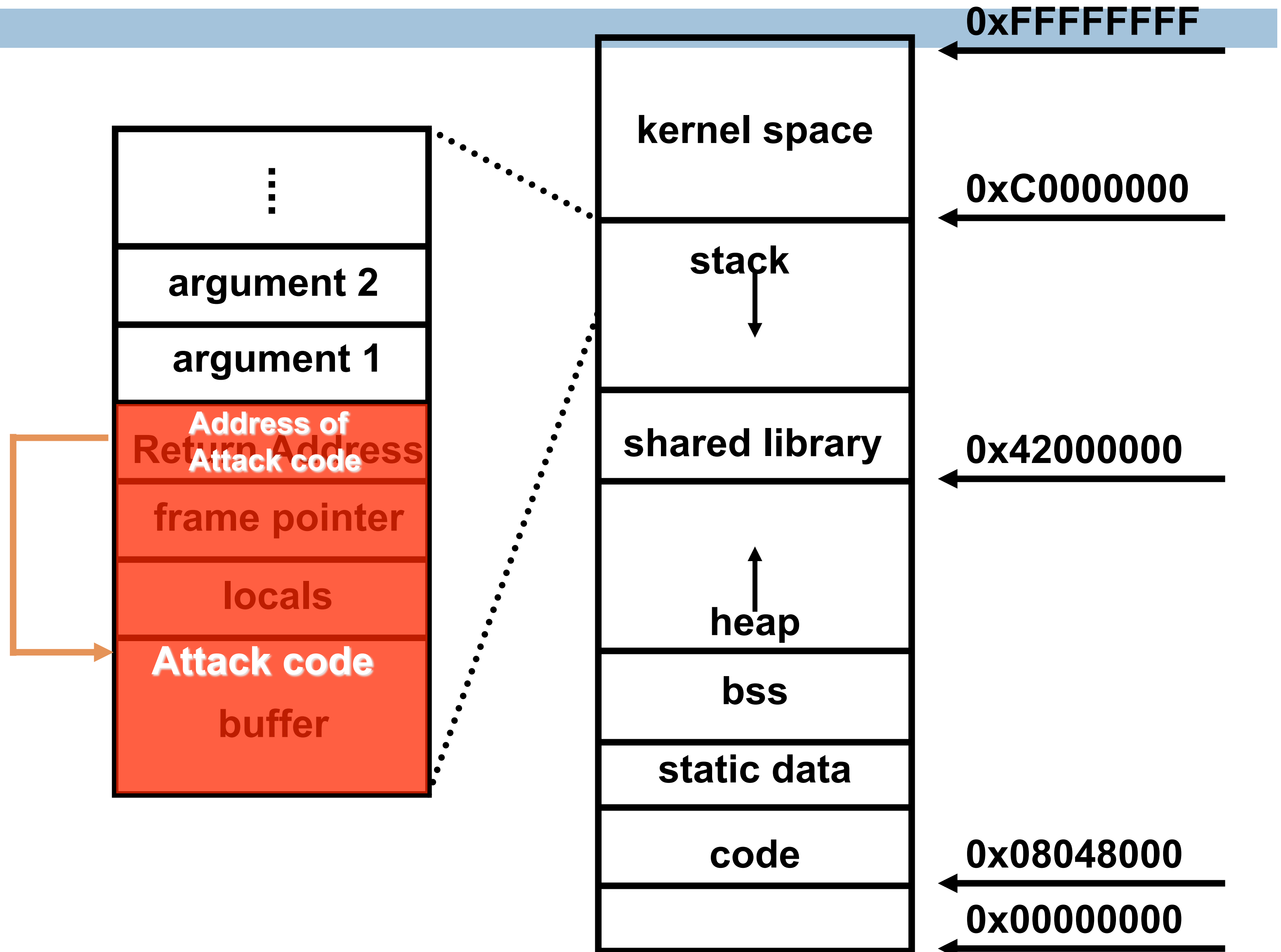
- Reliable Execution
  - Injected code vs. existing code
  - Absolute vs. relative address dependencies
- Complexity vs. Features

- Anatomy of Code-Injection Attacks
- Lab 3: Buffer Overflow

Buffer Overflow Recap

- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - **1:** Inject attack code into buffer
  - **2:** Redirect control flow to attack code
  - **3:** Execute attack code

Typical Address Space (Linux)



- To successfully launch a buffer overflow attack, two locations are important
  - The location of return address
  - The location of attack code