WILLIAM & MARY

CHARTERED 1693

CSCI 667: Concepts of Computer Security

Lecture 15

Prof. Adwait Nadkarni

Derived from slides by William Enck, Patrick McDaniel and Trent Jaeger

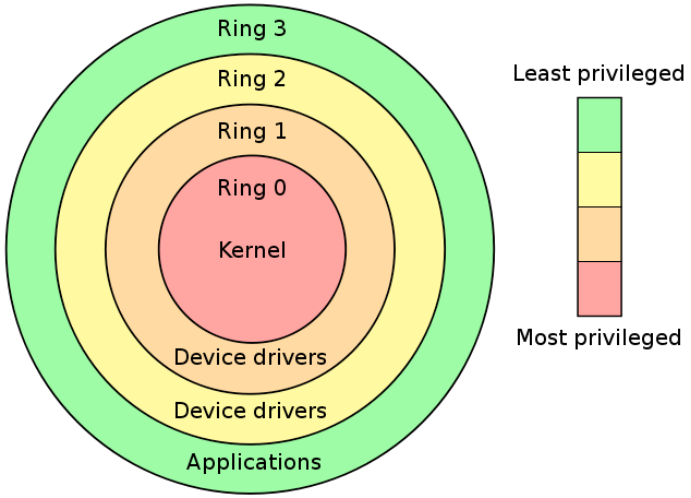
Announcements!

- **Research Plan due: Nov 14 (next Thursday)**
- **Project Status Presentations: Nov 26**
  - 10 minute talks
  - Focus on *status*, feedback
  - Do not expect finished projects
  - 1-5 bonus points

Multics (continued)

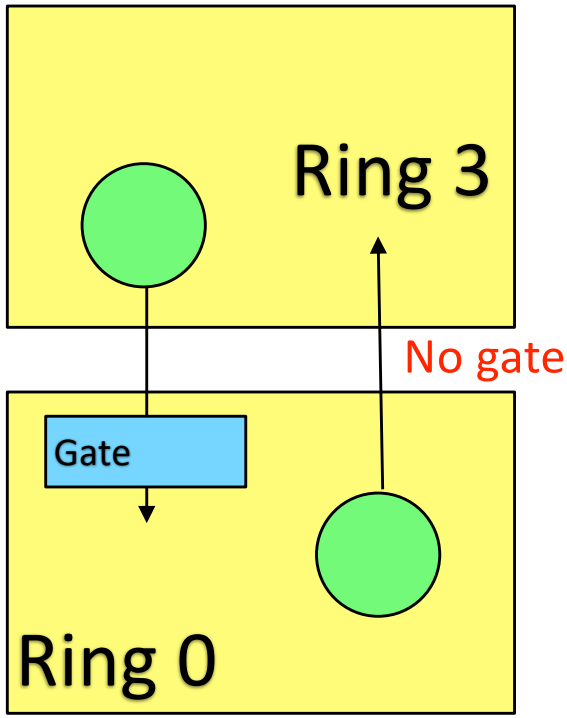
What Are Protection Rings?

- Coarse-grained, Hardware Protection Mechanism
- Boundary between Levels of Authority
  - Most privileged -- ring 0
  - *Monotonically less privileged above*
- Fundamental Purpose
  - Protect system integrity
    - Protect kernel from services
    - Protect services from apps
    - So on...



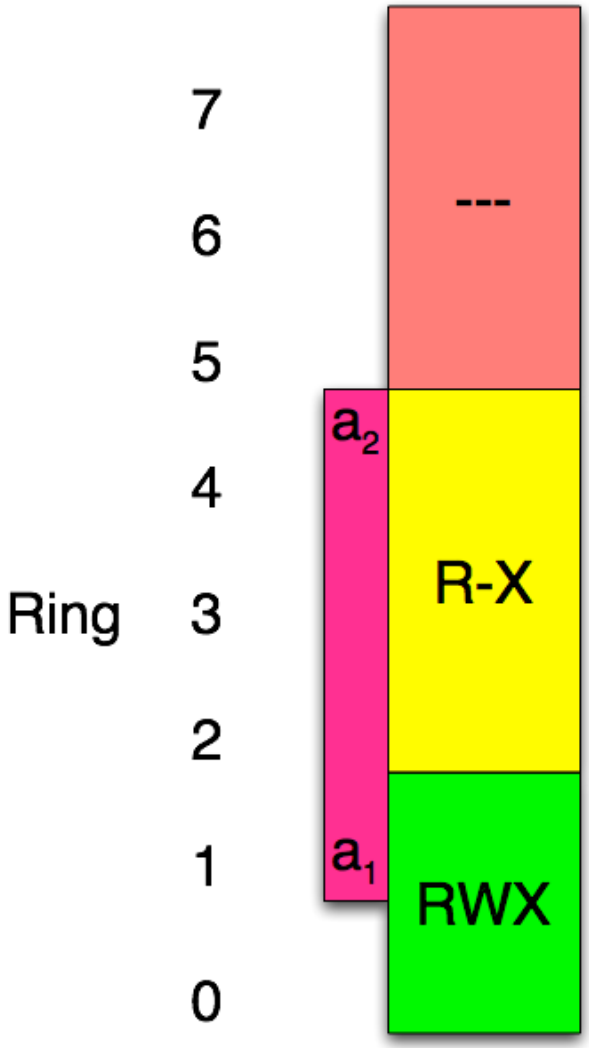
Protection Ring Rules

- Program cannot call code of *higher privilege* directly
  - Gate is a special memory address where lower-privilege code can call higher
    - Enables OS to control where applications call it (system calls)



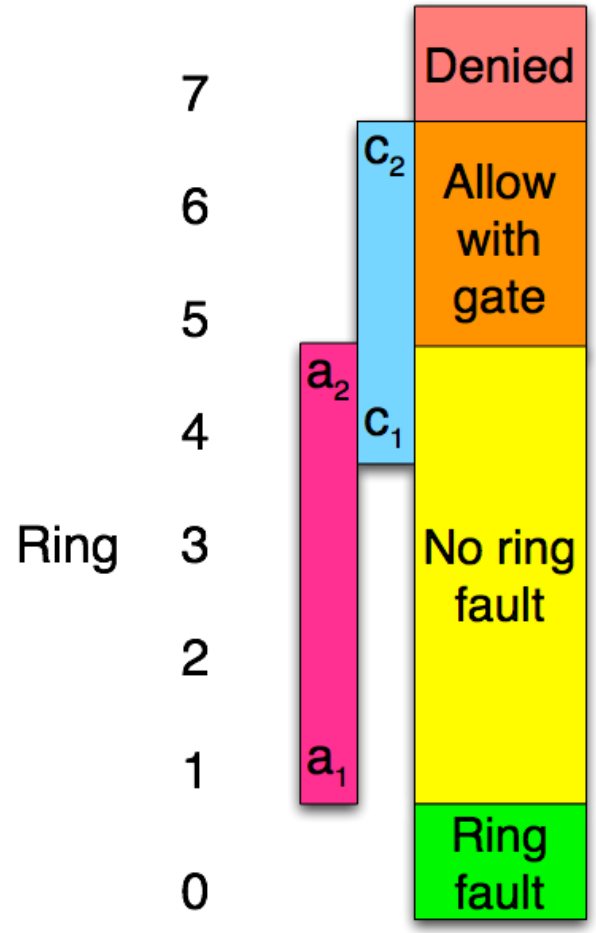
Multics Interpretation

- Kernel resides in ring 0
- Process runs in a ring r
  - Access based on current ring
- Process accesses data (segment)
  - Each data segment has an *access bracket*: (a1, a2)
    - a1 <= a2
  - Describes read and write access to segment
    - r is the current ring
    - r <= a1: access permitted
    - a1 < r <= a2: r and x permitted; w denied
    - a2 < r: all access denied



Multics Interpretation (cont'd)

- Also different procedure segments
  - with *call brackets*: (c1, c2), c1 <= c2
  - and access brackets (a1, a2)
  - The following must be true (a2 == c1)
  - Rights to execute code in a new procedure segment
    - r < a1: access permitted with ring-crossing fault
    - a1 <= r <= a2 = c1: access permitted and no fault
    - a2 < r <= c2: access permitted through a valid gate
    - c2 < r: access denied
- What's it mean?
  - case 1: ring-crossing fault changes procedure's ring
    - increases from r to a1
  - case 2: keep same ring number
  - case 3: gate checks args, decreases ring number
- Target code segment defines the new ring



Examples

- Process in ring 3 accesses data segment
  - access bracket: (2, 4)
  - What operations can be performed?
- Process in ring 5 accesses same data segment
  - What operations can be performed?
- Process in ring 5 accesses procedure segment
  - access bracket (2, 4)
  - call bracket (4, 6)
  - Can call be made?
  - How do we determine the new ring?
  - Can new procedure segment access the data segment above?

Multics Segments

- *Named segments* are protected by access control lists and MLS protections
  - Hierarchically arranged
  - Precursor to hierarchical file systems
- *Memory segment* access is controlled by hardware monitor
  - Multics hardware retrieves *segment descriptor word* (SDW)
  - Like a file descriptor
  - Based on rights in the SDW determines whether can access segment
- *Master mode* (like root) can override protections

- Access a directory or SDW on each instruction!

Multics Vulnerability Analysis

- Detailed security analysis covering
  - Hardware
  - Software
  - Procedural features (administration)
- Good news
  - Design for security
  - System language prevents buffer overflows
    - Defined buffer sizes
  - Hardware features prevent buffer overflows
    - Addressing off segment is an error
    - Stack grows up
  - System is much smaller than current UNIX systems

Vulnerabilities Found

- Hardware
  - Indirect addressing -- incomplete mediation
    - Check direct, but not indirect address
  - Mistaken modification introduced the error
- Software
  - Ring protection (done in software)
    - Argument validation was flawed
    - Certain type of pointer was handled incorrectly
  - Master mode transfer
    - For performance, run master mode program (signaler) in user ring
    - Development assumed trusted input to signaler -- bad combo
- Procedural
  - Trap door insertion goes undetected

Program Vulnerabilities

Programming

- Why do we write programs?
  - Function
- What functions do we enable via our programs?
  - Some we want -- some we don't need
  - Adversaries take advantage of such "hidden" function

A Simple Program

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

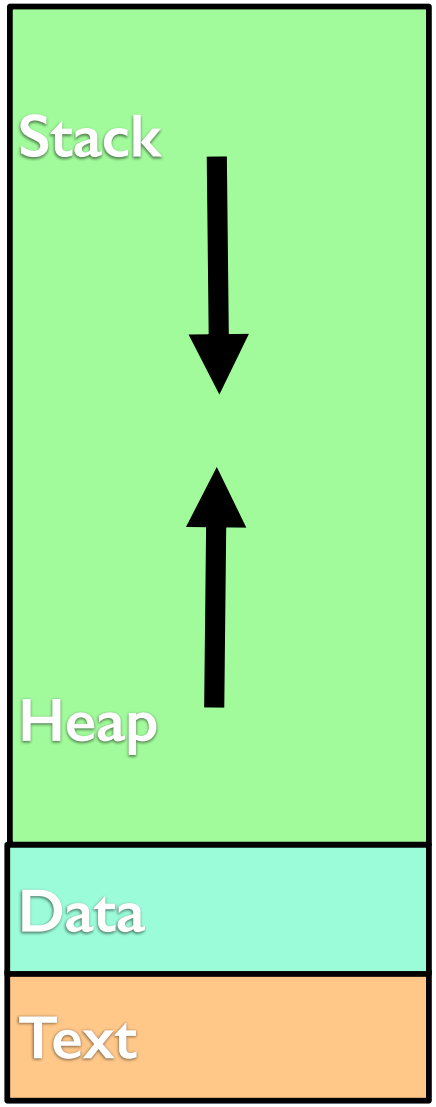
A Simple Program

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

What if packet is larger than 1000 bytes?

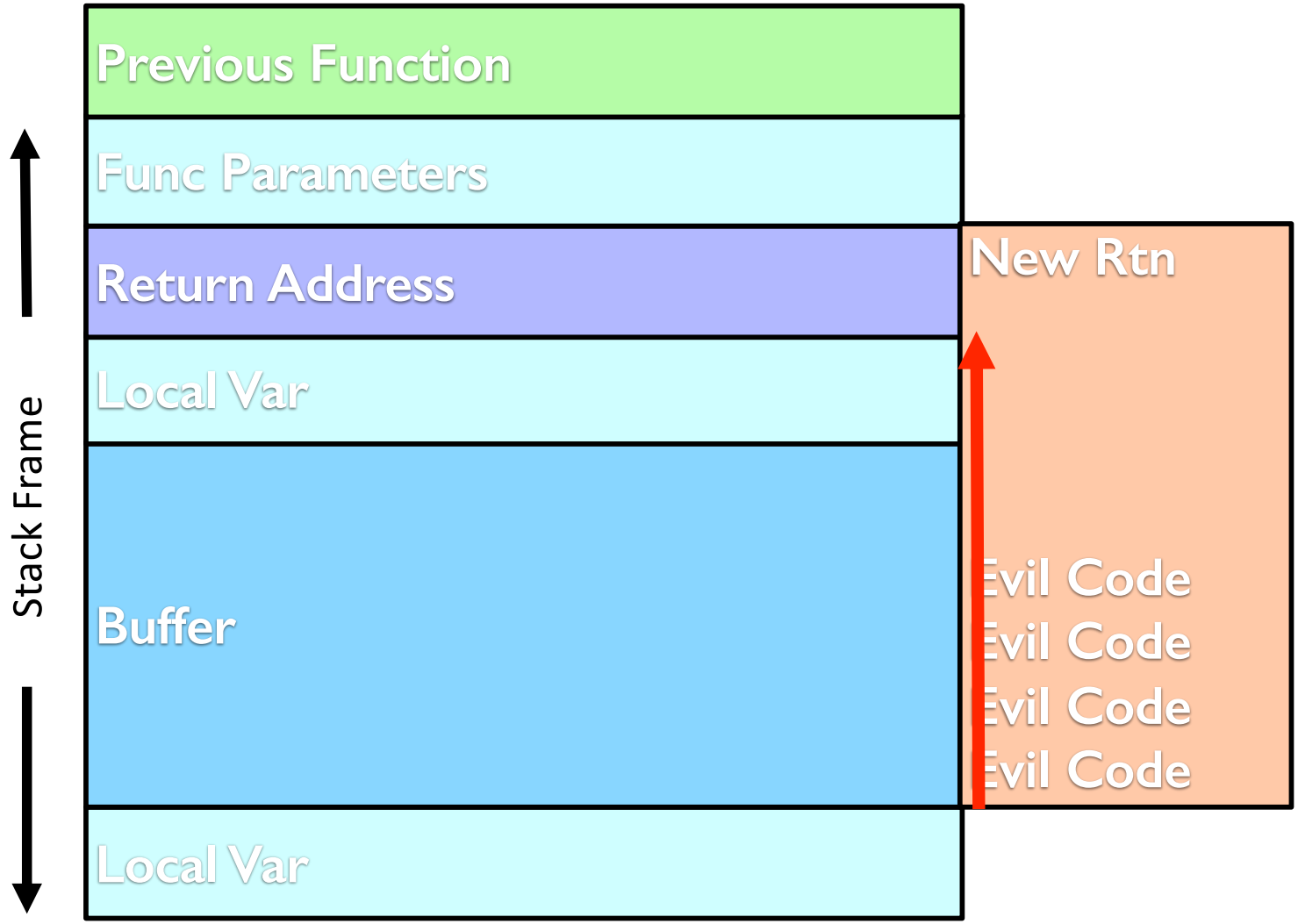
Address Space Layout



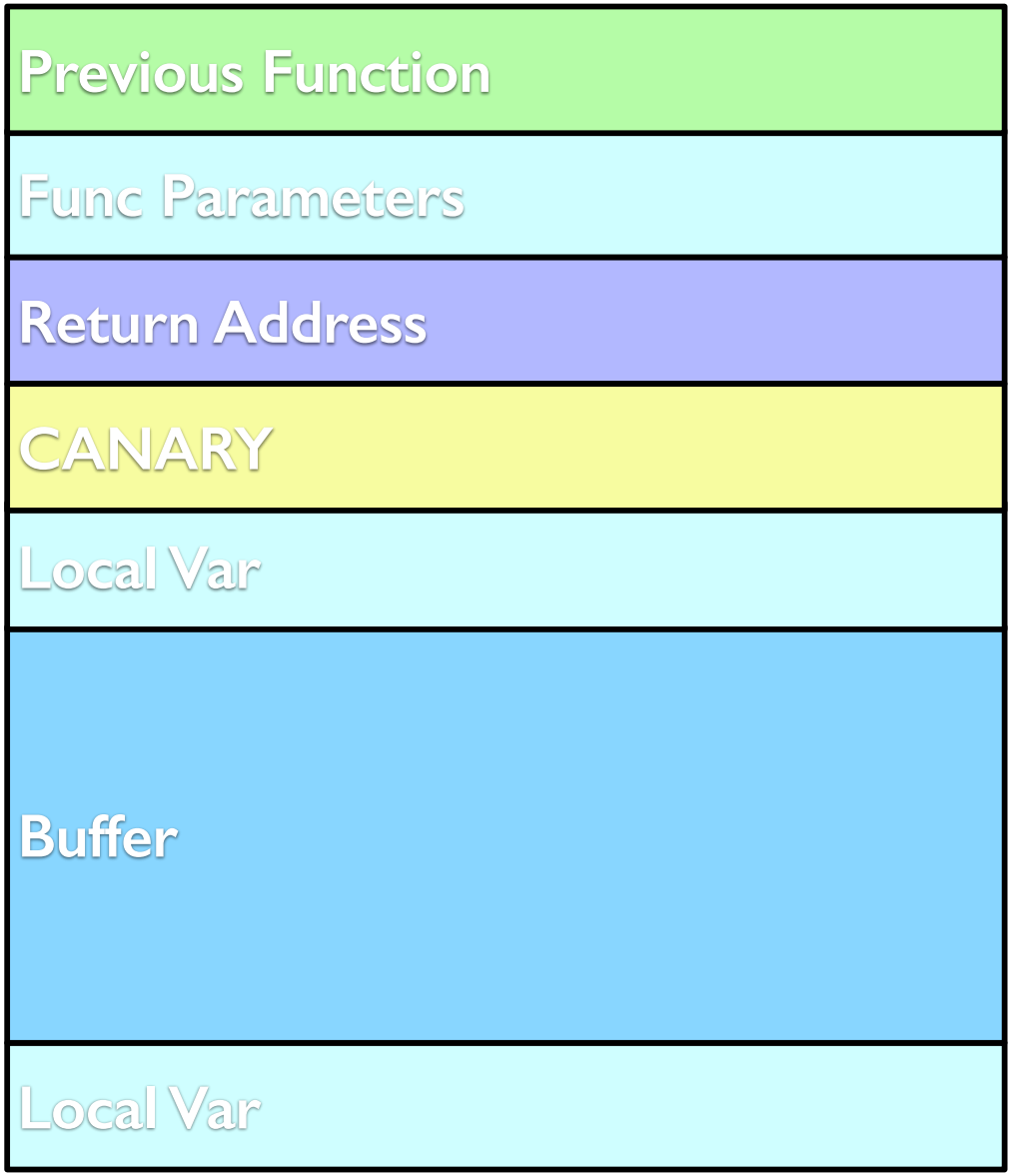
- Write beyond variable limit
  - Can write the without limits in some languages
- Can impact values
  - In heap, on stack, in data
- Can impact execution integrity
  - Can jump to arbitrary points in the program
    - Function pointers
    - Return addresses

Buffer Overflow

- How it works



Buffer Overflow Defense



- "Canary" on the stack
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped
- Are we done?

A Simple Program

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
   if (Authenticate(packet))
     authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

What if packet is only 1004 bytes?

Overflow of Local Variables

- Don't need to modify return address
  - Local variables may affect control
- What kinds of local variables would impact control?
  - Ones used in conditionals (example)
  - Function pointers
- What can you do to prevent that?

A Simple Program

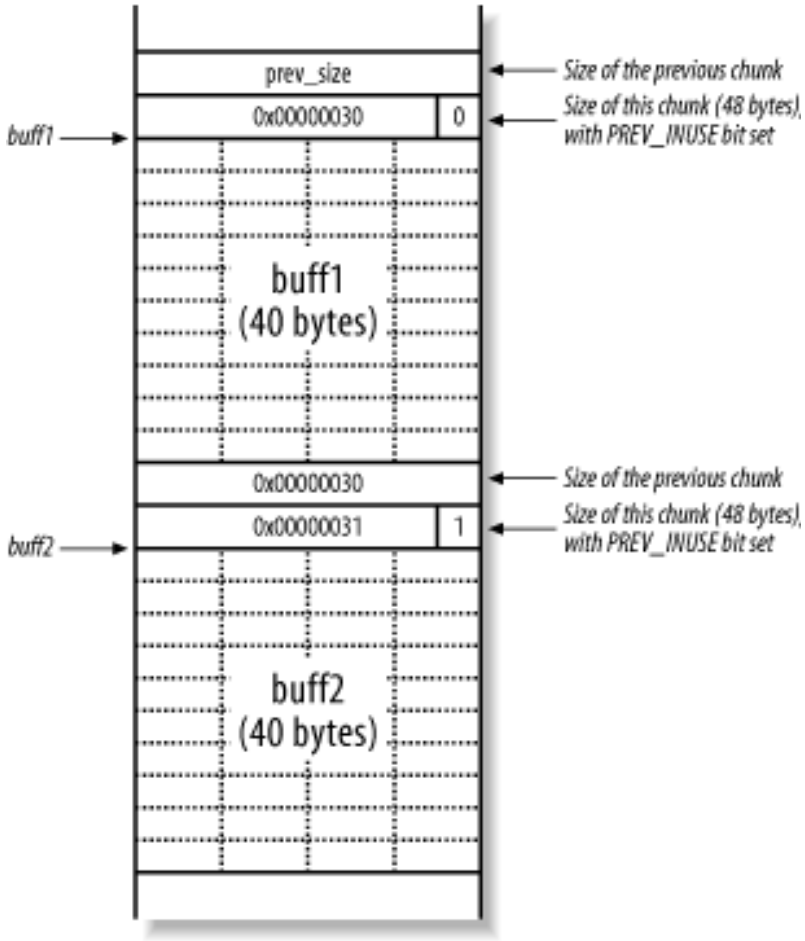
```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

What if we allocate the packet buffer on the heap?

Heap Overflow

- Overflows may occur on the heap also
  - Heap has data regions and metadata
- Attack
  - Write over heap with target address (heap spraying)
  - Hope that victim uses an overwritten function pointer before program crashes



Another Simple Program

```
int size = BASE_SIZE;
char *packet = (char *)malloc(1000);
char *buf = (char *)malloc(1000+BASE_SIZE);

 strcpy(buf, FILE_PREFIX);
 size += PacketRead(packet);
 if ( size < sizeof(buf)) {
   strcat(buf, packet);
   fd = open(buf);
 }
```

Any problem with this conditional check?

Integer Overflow

- Signed variables represent positive and negative values
  - Consider an 8-bit integer: -128 to 127
  - Weird math: *127+1 = ???*
- This results in some strange behaviors
  - **`size += PacketRead(packet)`**
    - What is the possible value of size?
  - **`if ( size < sizeof(buf)) {`**
    - What is the possible result of this condition?
- How do we prevent these errors?

```
qsee_not_in_region(list, start,
start+size);
...
int qsee_not_in_region(void
*list, long start, long end)
{
        if (end < start)
        { tmp = start; start = end;
        end = tmp; }
        //    Perform validation ...
}
```

https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf

A Simple Program

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
if (authenticated)
   ProcessQuery("Select", partof(packet));
```

Any problem with this query request?

Parsing Errors

- Have to be sure that user input can only be used for expected function
  - *SQL injection*: user provides a substring for an SQL query that changes the query entirely (e.g., add SQL operations to query processing)

```
SELECT fieldlist FROM table
WHERE field = 'anything' OR 'x'='x';
```

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

  - Goal: format all user input into expected types and ranges of values
    - Integers within range
    - Strings with expected punctuation, range of values

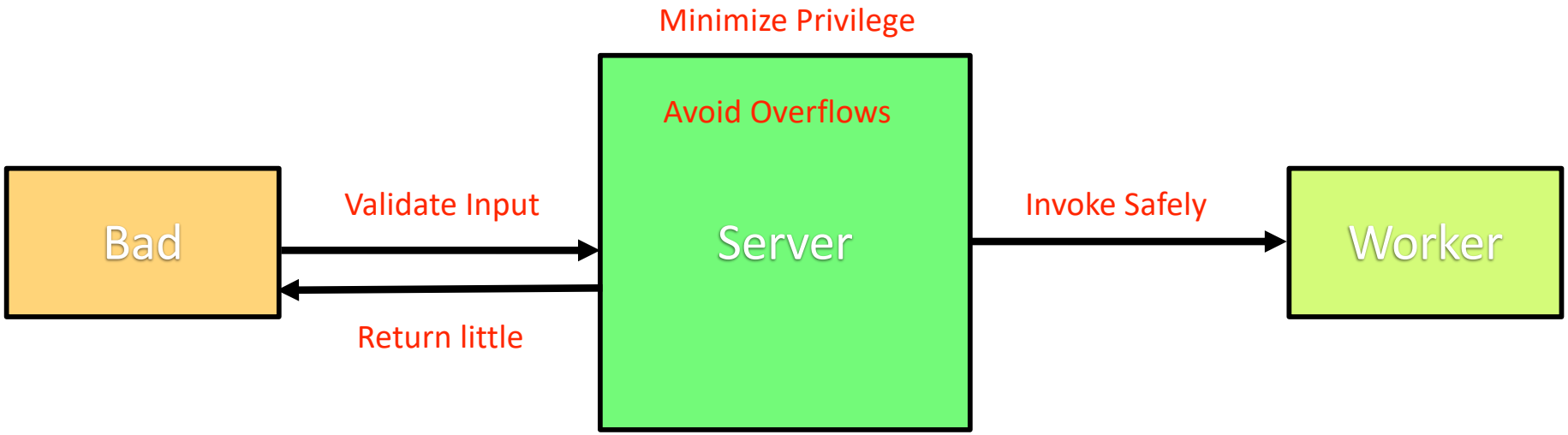
- Many scripting languages convert data between types automatically -- are not *type-safe* -- so must be extra careful

Character Strings

- String formats
  - Unicode
    - ASCII -- 0x00 -- 0x7F (files and strings which contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.)
    - non-ASCII -- 0x80 -- 0xF7
    - Also, multi-byte formats
  - Decoding is a challenge
    - URL: [IPaddr]/scripts/..%c0%af../winnt/system32
    - Decodes to /winnt/system32
  - Markus Kuhn's page on Unicode resources for Linux
    - www.cl.cam.ac.uk/~mgk25/unicode.html

Secure Input Handling

- David Wheeler's Secure Programming for Linux and UNIX
  - Validate all input; Only execute application-defined inputs!
  - Avoid the various overflows
  - Minimize process privileges
  - Carefully invoke other resources
  - Send information back carefully



The End