CMPSC 447
Final Review

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Final

- Format (114 points)
  - True/False
    - 8 questions – 16 points
  - Short answer – word/phrase to sentence or two
    - 8 questions – 36 points
  - Long answer – like essay-ish
    - 4 questions – 32 points
  - Constructions – how
    - 3 questions (multi-part) – 30 points
  - Time should be less of an issue, but be careful

Defenses Quiz

- #1 - When a stack canary is overwritten, then the program check (immediately) terminates the process execution.

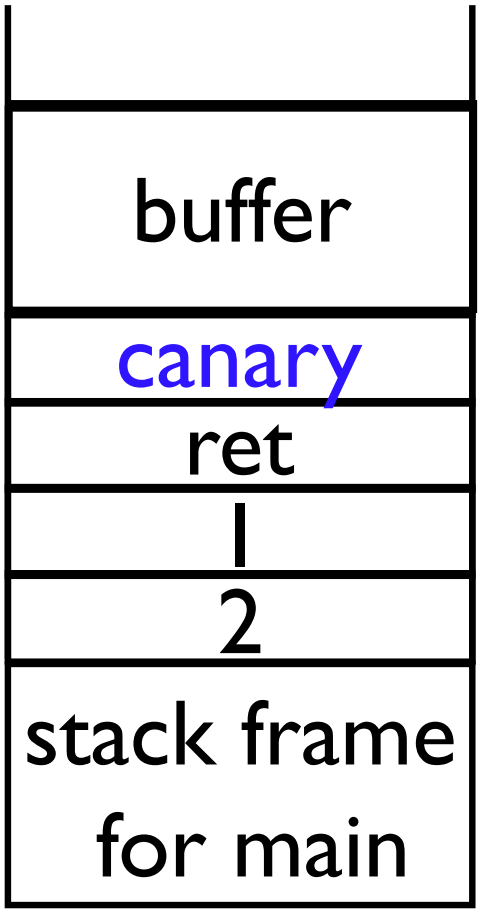
- True/False

Defenses Quiz

- #1 - When a stack canary is overwritten, then the program check (immediately) terminates the process execution.

- True/False

Stack Canary Defense



- Place a "canary" value on the stack to detect attempted overwrites of the return address
- Canary value is randomized
- And checked prior to any return
- How does this prevent overflows from exploiting the return address?

Defenses Quiz

- #2 - The randomness of Address Space Layout Randomization (ASLR) is restricted by the relative positions of the individual memory regions (stack, heap, etc.).
  - True/False

Defenses Quiz

- #2 - The randomness of Address Space Layout Randomization (ASLR) is restricted by the relative positions of the individual memory regions (stack, heap, etc.).
  - True/False

ASLR

- Create a memory segment
  - Heap
  - Stack
  - Code (Library)
- Compute (randomize) the base address
  - High order bits – fixed – segment needs to be placed in the expected relative position
  - Some middle bits – random – this is where ASLR is applied
  - Low order bits – align – must be at least page aligned
- Limits the "entropy" of the randomization
  - Number of possible locations - 2^n where n is entropy in "bits"

Defenses Quiz

- #3 - Coarse-grained Control-Flow Integrity (CFI) restricts the allowed targets of an indirect call site to the following set of program code locations.
  - The targets of this call site in the CFG
  - The functions with the same type signature as the call site
  - Any caller of the function with this call site
  - Any code instruction
  - The start of any function

Defenses Quiz

- #3 - Coarse-grained Control-Flow Integrity (CFI) restricts the allowed targets of an indirect call site to the following set of program code locations.
  - The targets of this call site in the CFG
  - The functions with the same type signature as the call site
  - Any caller of the function with this call site
  - Any code instruction
  - The start of any function

CFI Policies

- Coarse-grained Policy
  - Check if the targets of indirect control transfers are valid
  - Requires decoding the trace packets to find each target
- Fine-grained Policy
  - Check if the source and destination are a legitimate pair
  - Requires control-flow recovery to identify source
- Shadow Stack
  - Check that a function can only return to its caller (instruction after the call site)
  - Check if an indirect control transfer is legitimate based on the program state (e.g., shadow stack)

Defenses Quiz

- #4 - The return address address of a function is most accurately restricted to the set of legal callers by the following.
  - Shadow Stack
  - Coarse-grained CFI
  - Fine-grained CFI
  - Stack Canary
  - CFG

Defenses Quiz

- #4 - The return address address of a function is most accurately restricted to the set of legal callers by the following.
  - Shadow Stack
  - Coarse-grained CFI
  - Fine-grained CFI
  - Stack Canary
  - CFG

Defenses Quiz

- You can limit an load/store instruction to the range of addresses between 0x1400 and 0x14FF by "masking" the load/store address X using "X & _____" followed by "X | _____”

Defenses Quiz

- You can limit an load/store instruction to the range of addresses between 0x1400 and 0x14FF by "masking" the load/store address X using "X & _____" followed by "X | _____"
  - X & 0x00FF
  - X | 0x1400

Defenses Quiz

- #6 – If some of your program's code is restricted to a limited memory region by Software-Fault Isolation, the code can only invoke a function outside that region by jumping to an address defined by the following.
  - Sandbox
  - Segment
  - Trampoline
  - Stub
  - Mask

Defenses Quiz

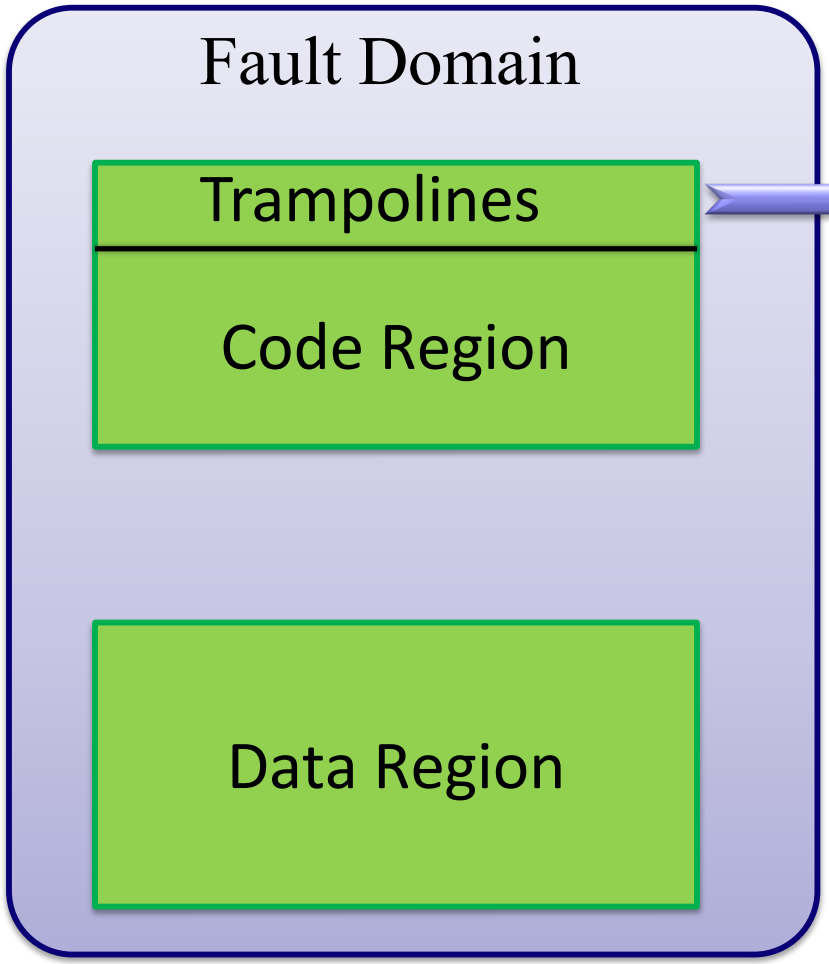
- #6 – If some of your program's code is restricted to a limited memory region by Software-Fault Isolation, the code can only invoke a function outside that region by jumping to an address defined by the following.
  - Sandbox
  - Segment
  - Trampoline
  - Stub
  - Mask

Jumping Outside of Domain

PENNSTATE 1855

- Sometimes need to invoke code outside of the domain
  - For system calls; for communication with other domains
  - Danger: Cannot allow untrusted code to invoke code outside of the fault domain arbitrarily
- Idea:
  - Insert a jump table into the (immutable) code region
  - Each entry is a control transfer instruction whose target address is a legal entry point outside of the domain

A Fixed Jumptable (Trampoline)



stubs to trusted routines

- For example
  - Trampolines for system calls: fopen; fread; …
  - Trampolines for communication with other fault domains

Trusted Stubs

PENNSTATE 1855

- Stubs are outside of the fault domain
  - Why?
- Stubs can implement security checks
  - E.g., can restrict fopen to open files only in a particular directory
  - Or can disallow fopen completely
    - Just not install a jump table entry for it
  - It can implement system call interposition

Defenses Quiz

- #7 – In the code below, why is the "key" not leaked from the function encrypt?

- ```
  char* cipher;
  char* key;
  ```

- ```
  void encrypt(char *plain, int n) {
  cipher = (char*)malloc(n);
  for (i = 0; i < n; i++) { cipher[i] = plain[i] ^ key[I]; }
  }
  ```

- ```
  void main (){
  char plaintext[1024];
  scanf("%s",plaintext);
  encrypt(plaintext,strlen(plaintext));
  ...
  }
  ```
  ```

Defenses Quiz

- #7 – In the code below, why is the "key" not leaked from the function encrypt?

- ```
  char* cipher;
  char* key;
  ```

- ```
  void encrypt(char *plain, int n) {
  cipher = (char*)malloc(n);
  for (i = 0; i < n; i++) { cipher[i] = plain[i] ^ key[I]; }
  }
  ```

- ```
  void main (){
  char plaintext[1024];
  scanf("%s",plaintext);
  encrypt(plaintext,strlen(plaintext));
  ...
  }
  ```

- Encryption as a declassifier
  ```

OpenSSH Privilege Separation

PENNSTATE 1855

- What parts of code need access to sensitive data and privileges in OpenSSH?
  - Code that needs access to root privileges
    - to change UID of child process (integrity)
  - Code that needs access to critical secrets
    - For setting up secure channels and password authentication (secrecy)
- How would you privilege separate these functionalities from the rest of OpenSSH?

OpenSSH Privilege Separation

- How OpenSSH looks after privilege separation

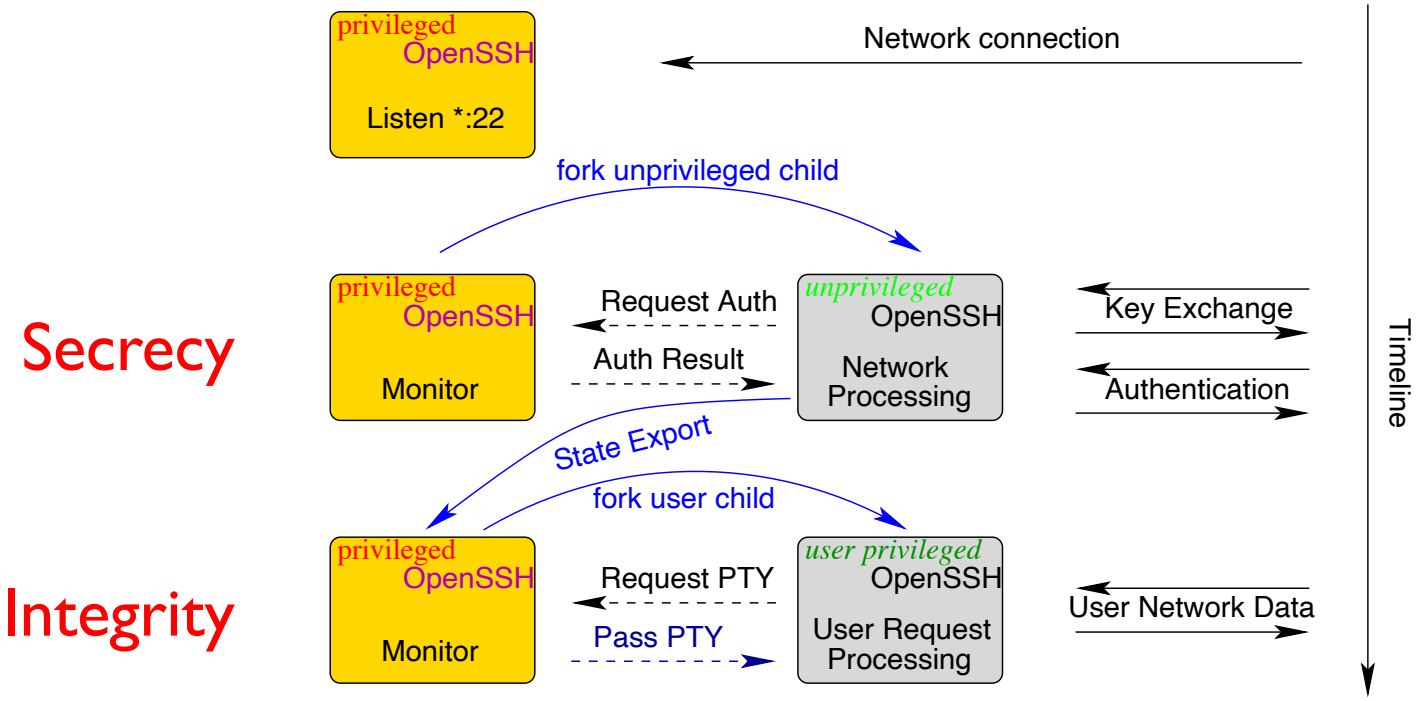


Figure 4: Overview of privilege separation in OpenSSH. An unprivileged slave processes all network communication. It must ask the monitor to perform any operation that requires privileges.

Separation Issues

- Information Flow Issues
  - Secrecy
    - Secret component must return authentication result
    - Filter secrets from the response (declassify)
  - Integrity
    - High integrity component must receive input
    - Validate integrity of untrusted inputs (endorsement)
  - Both
    - In many cases the secret data is also high integrity
    - What then?

Defenses Quiz

- #8 – When enforcing CFI using process traces collected using Intel Processor Trace, CFI is not enforced until a system call is made because...
  - Process traces only collect system calls
  - A system call stops all process threads
  - The OS cannot process traces completely until a system call occurs
  - Only a system call can impact the system
  - System calls are necessary to violate control flow

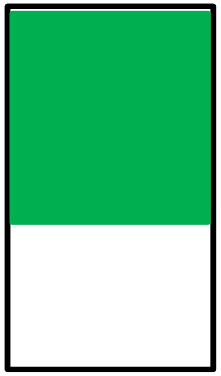
Defenses Quiz

- #8 – When enforcing CFI using process traces collected using Intel Processor Trace, CFI is not enforced until a system call is made because… (from a security perspective)
  - Process traces only collect system calls
  - A system call stops all process threads
  - The OS cannot process traces completely until a system call occurs
  - Only a system call can impact the system
  - System calls are necessary to violate control flow

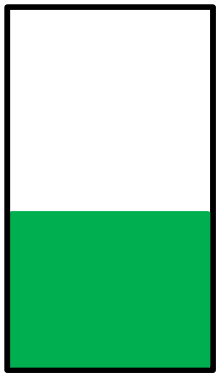
System Overview

User Space

Kernel Space



SYSCALL



Defenses Quiz

- #9 – Enclaves protect a program (being run in an enclave from attacks from ...
  - Other threads in the same enclave
  - Other enclaves
  - Other processes outside of enclaves
  - The processor
  - The operating system

Defenses Quiz

- #9 – Enclaves protect a program (being run in an enclave from attacks from ...
  - Other threads in the same enclave
  - Other enclaves
  - Other processes outside of enclaves
  - The processor
  - The operating system
- Anything but what is in the enclave and the processor

Analysis Quiz

- #1 - A vulnerability is a _____ that is _____ to an adversary who can _____ it (Ans1).
  - Three answers

Analysis Quiz

- #1 - A vulnerability is a _____ that is _____ to an adversary who can _____ it (Ans1).
  - Flaw
  - Accessible
  - Exploit

Analysis Quiz

- #2 - Redo Question 14 of the midterm

```
1: int a;                              // size input set in line 5
2: char adv_input_2[SIZE];             // SIZE is a constant
3: char b[SIZE];
4:
5: a=adv_input_1;
6: if (a<SIZE) {
7:         read(&adv_input_2, a);          // read(char **dest, size_t size);
8:         strlcpy(b, adv_input_2, a); // strlcpy(char *dest, char *src,
size_t size);
9:         send(b, a);       // send(char *msg, size_t size); to adversary
10: }
11: return;
```

Analysis Quiz

- #3 - Use static analysis - via abstract interpretation - to detect that an adversary-controlled input is used to define an invalid size variable "a" in Question 14 of the midterm.

```
1: int a;                                  // size input set in line 5
2: char adv_input_2[SIZE];                 // SIZE is a constant
3: char b[SIZE];
4:
5: a=adv_input_1;
6: if (a<SIZE) {
7:         read(&adv_input_2, a);              // read(char **dest, size_t size);
8:         strlcpy(b, adv_input_2, a); // strlcpy(char *dest, char *src,
size t size);
9:         send(b, a);      // send(char *msg, size t size); to adversary
10: }
11: return;
```

Abstract Interpretation

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For an expression, $c = a * b$
  - If a has a descriptor *pos*
  - And b has a descriptor *neg*
- What is the descriptor for c after that instruction?
  - Need a rule to combine two descriptors for each op
  - For '*' use rule for multiplication of pos and neg
- How might this help?

Analysis Quiz

- #4 – For the code below

```
F1() { return; }

F2() { F1(); }

FA(fptr) { fptr(); }
```

- The most accurate CFI policy for returns from F1 given that it is assigned to "fptr" is:

- F2 / FA / F1 and FA / F1 and F2 and FA / F2 and FA

Analysis Quiz

- #4 – For the code below

```
F1() { return; }

F2() { F1(); }

FA(fptr) { fptr(); }
```

- The most accurate CFI policy for returns from F1 given that it is assigned to "fptr" is:

- F2 / FA / F1 and FA / F1 and F2 and FA / F2 and FA

Analysis Quiz

- #5 – What is the best way to enable fuzz testing to find inputs to take the "true" path of a conditional "(x == 0xffff2345)"?
  - Unsound Static Analysis
  - Sound Static Analysis
  - Generational Fuzzing
  - Symbolic Execution
  - Mutational Fuzzing

Analysis Quiz

- #5 – What is the best way to enable fuzz testing to find inputs to take the "true" path of a conditional "(x == 0xffff2345)"?
  - Unsound Static Analysis
  - Sound Static Analysis
  - Generational Fuzzing
  - Symbolic Execution
  - Mutational Fuzzing

Helping Fuzzing

```
x = int(input())
if x > 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

42 ⇒ "You lose!"

3 ⇒ "You lose!"

.....

57 ⇒ "You lose!"

Fuzzing vs. Symbolic Exec

```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
```

Fuzzing Wins

```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic Execution Wins

Analysis Quiz

- #7 – What are the path constraints for the following code to "you win"?

```
if (x < 10) {
      if (x > 0)
              return "you lose";
      else
              return "you win"
}
else
     return "you lose"
```

- 10 > x > 0
- x > 0
- x < 10
- x >= 10
- x < 0

Analysis Quiz

- #7 – What are the path constraints for the following code to "you win"?

```
if (x < 10) {
      if (x > 0)
              return "you lose";
      else
              return "you win"
}
else
     return "you lose"
```

- 10 > x > 0
- x > 0
- x < 10
- x >= 10
- x < 0

Analysis Quiz

- #8 - Suppose your program has 2 systems calls, consisting of (1) open a file and (2) read its data. If an adversary controls both the file contents and the directory in which the file is stored, both of these system calls are part of the program's attack surface.

- True/False

Analysis Quiz

- #8 - Suppose your program has 2 systems calls, consisting of (1) open a file and (2) read its data. If an adversary controls both the file contents and the directory in which the file is stored, both of these system calls are part of the program's attack surface.

- True/False

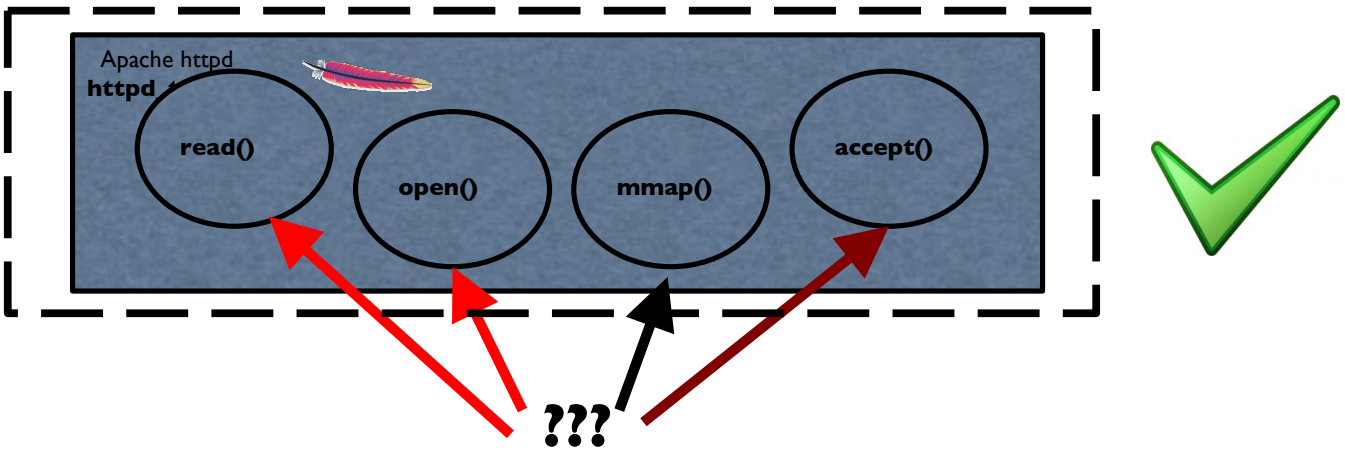
Filesystem Attacks

PENNSTATE 1855

- What is the threat that enables link traversal and file squatting attacks?
  - Common to both
- In both cases, the adversary has write permission to a directory that a victim uses in name resolution
  - Could be any directory used in resolution, not just the last one
  - Enables the adversary to plant links and/or files

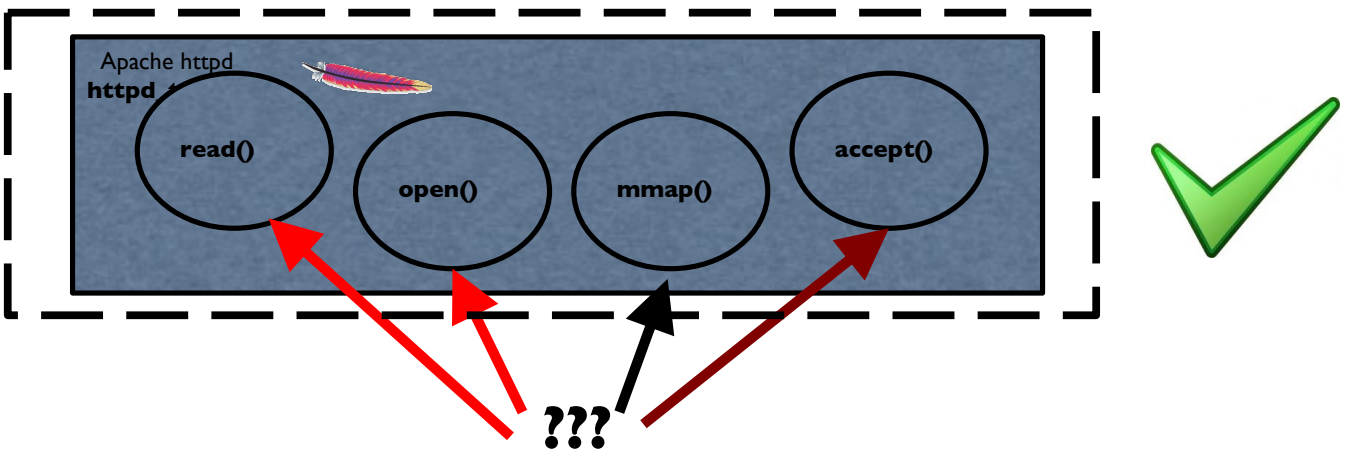
Attack Surface

- Insight: Only a small fraction system calls expect to use adversary-controlled input
  - Any new attack surface is often the source of vulnerabilities



Attack Surface

- Insight: Only a small fraction system calls expect to use adversary-controlled input
  - Attack surface set of program entry points (system call instances in your program) accessible to an adversary



  - Know this definition!

From Midterm Review

- At least
  - Confused Deputy
  - Stack and Heap Exploits
  - Safe String API use
  - Memory error types and associated attacks
- No ROP gadgets or MITRE ATT&CK

Take Away

- Review for final from the quiz questions and their answers
- Scope of exam includes these questions
  - And a little more
    - Review midterm too
  - Consider combination between memory safety and defenses/analyses
    - E.g., Spatial safety and static analysis
- Think about variants of these questions to give yourself a broader understanding
- Good luck!