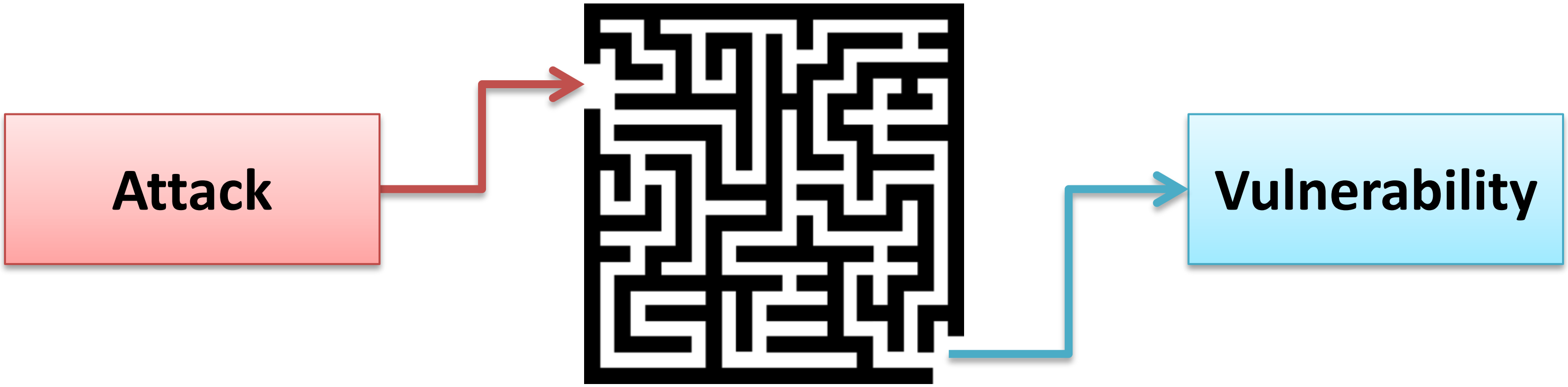
Pinpointing Vulnerabilities

Yue Chen, Mustakimur Khandaker, Zhi Wang
Florida State University

Question

- When an **attack** is detected, how to locate the underlying **vulnerability**?



Example

```
1 int main(int argc, char *argv[])
2 {
3       char buf[16];
4       strcpy(buf, argv[1]);
5       printf("%s\n", buf);
6       return 0;
7 }
```

Root Cause

Symptom

- A control-flow violation is detected at line **6**.
- The vulnerability lies at line **4** (buffer overflow).

Attack Detection v.s. Vulnerability Locating

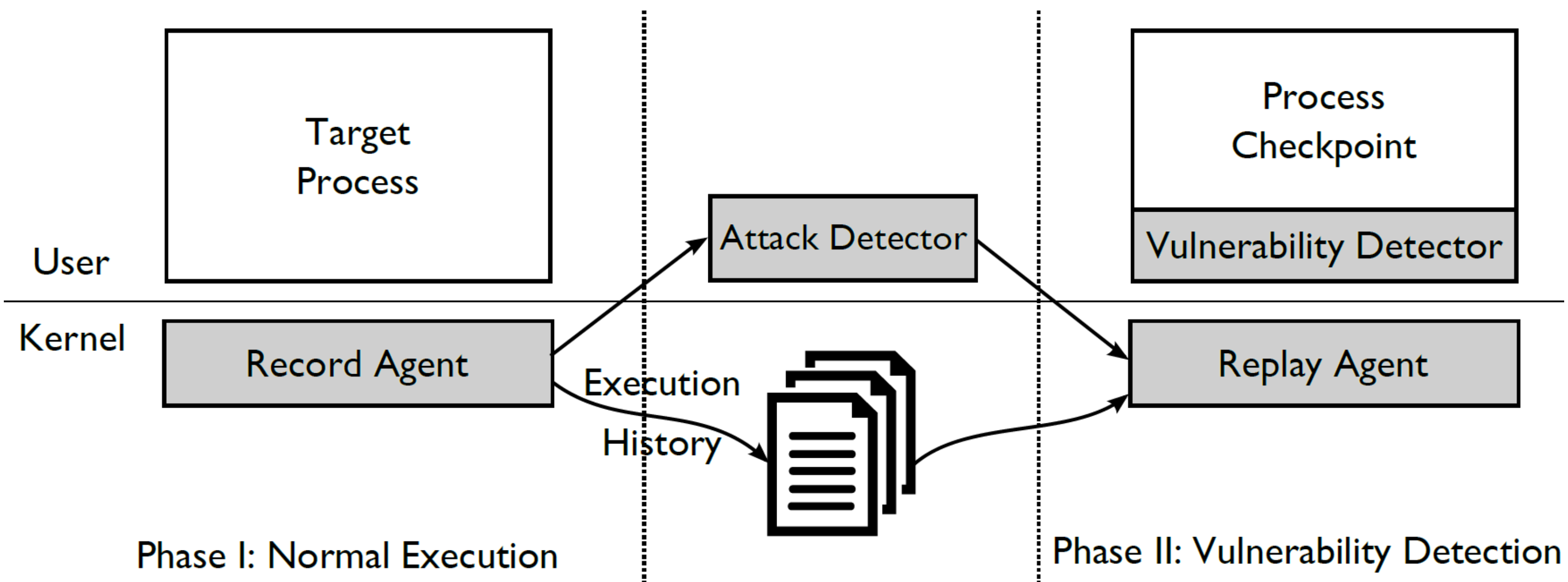
- **Control-flow Integrity (CFI)**
  - Detect the control-flow graph violation (e.g., on function returns)
- **Taint Analysis**
  - Detect tainted data being loaded to PC
- **System Call Interposition**
  - Detect abnormal syscalls made by the payload

Manifestation of attack rarely coincides with the vulnerabilities

Ravel – Three Components

- **Online** attack detector
- Record & replay with instrumentation
- **Offline** vulnerability locator

RAVEL:
Root-cause
Analysis of
Vulnerabilities from
Exploitation
Log



Ravel – Strengths

1. Reliably reproduce real-world attacks in the lab environment
2. Low online performance overhead
   – Locating vulnerabilities is time-consuming
3. Extensible:
   – New attack detection and vulnerability locating techniques can be easily integrated
   – (already support a variety of vulnerability locating techniques)

Attack Detection

- Ravel uses existing attack detection methods
  - Program crash (or other exceptions)
  - Abnormal system calls (sequence/arguments)
  - Control-flow integrity violation (to be included)
- New methods can be easily adopted by Ravel

Record & Replay

- What to record & replay?
  - All the non-deterministic inputs (e.g., network packets)
- Where to record & replay?
  - Application interface
  - Library interface
  - Virtual machine interface
  - System call interface

Record & Replay

- What to record & replay?
  - All the non-deterministic inputs (e.g., network packets)
- Where to record & replay?
  - Application interface
  - Library interface
  - Virtual machine interface
  - System call interface

More ***robust*** against attacks, with low cost

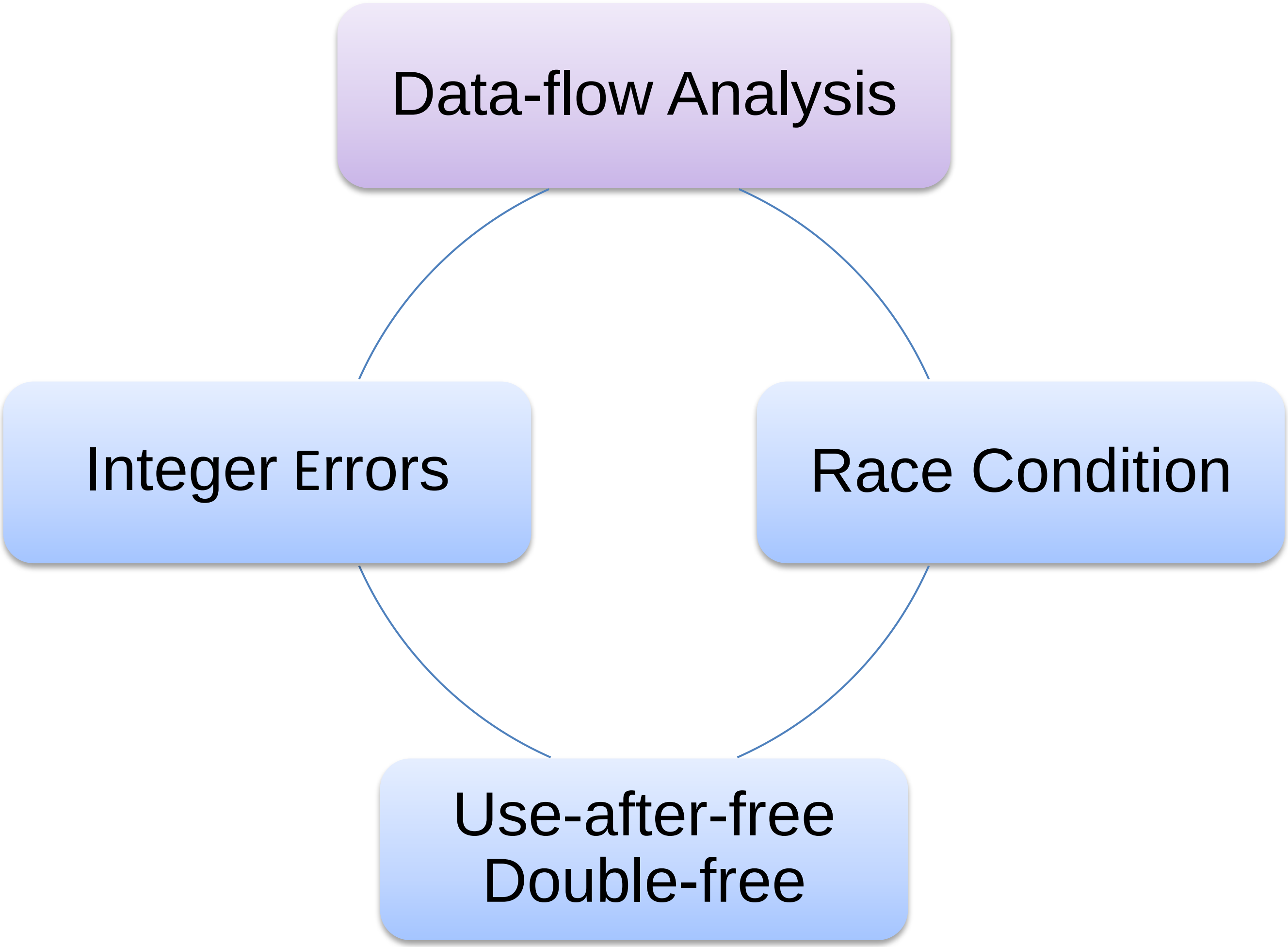
Record

- System call return values
- Userspace data structures modified by syscalls
- Data copied from kernel to userspace
- Asynchronous signals
- Special instructions (e.g., RDTSC)
- Synchronization primitives

Replay with Instrumentation

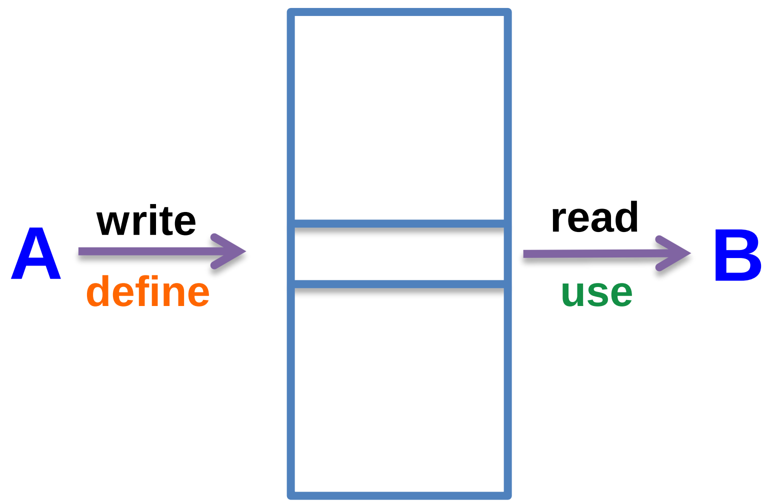
- Some syscalls replayed without real execution
  - e.g., gettimeofday
- Some syscalls need to be re-executed
  - e.g., mmap
- Replay under a binary translation (BT) engine
  - BT collects detailed memory accesses by the target
  - Replay distinguishes syscalls made by the target from those made by BT

Vulnerability Locator



Data-flow Analysis

- Analyze **def**-**use** relations between instructions
- **Define**: writes to a memory address
- **Use**: reads from a memory address



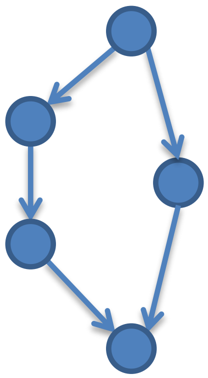
Data-flow Analysis

- Analyze **def**-**use** relations between instructions
- **Define**: writes to a memory address
- **Use**: reads from a memory address

$$A \longrightarrow B$$

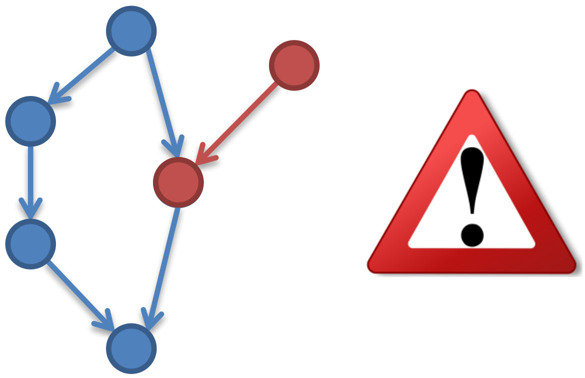
Data-flow Analysis

- Precompute a data-flow graph (DFG)
  - DFG: the **valid** def-use relations in the program
  - Our prototype uses dynamic analysis



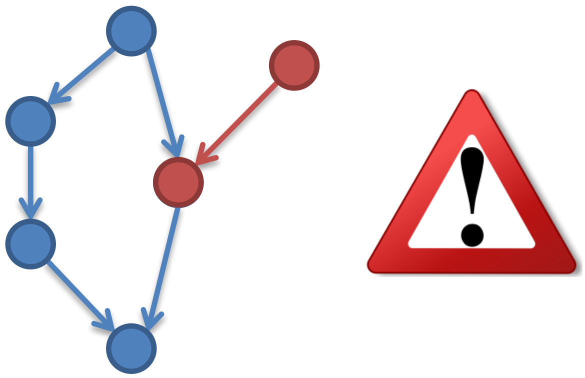
Data-flow Analysis

- Precompute a data-flow graph (DFG)
  - DFG: the **valid** def-use relations in the program
  - Our prototype uses dynamic analysis
  - Extra relations regarded as **violations**



Data-flow Analysis

- Precompute a data-flow graph (DFG)
  - DFG: the **valid** def-use relations in the program
  - Our prototype uses dynamic analysis
  - Extra relations regarded as **violations**
- Violation to DFG indicates the vulnerability location
  - It could be the **def** or the **use**, but which one?
  - Refine the results with heuristics

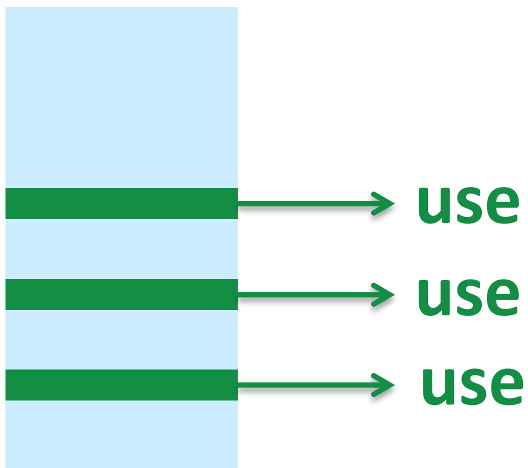


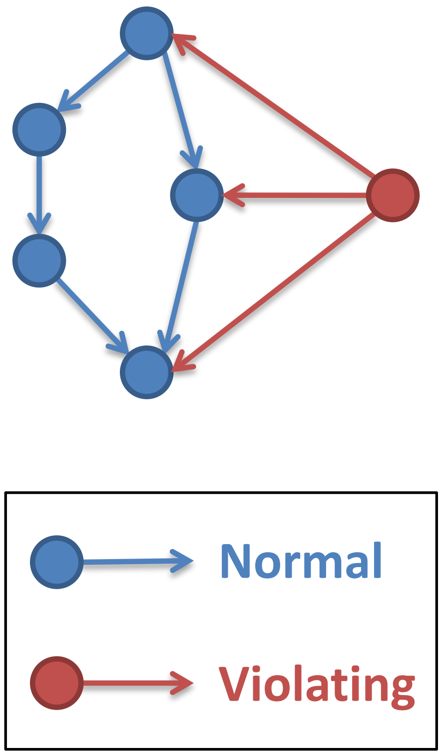
Data-flow Analysis Heuristics

- One **def**, many **uses**:

  def is closer to the vulnerability

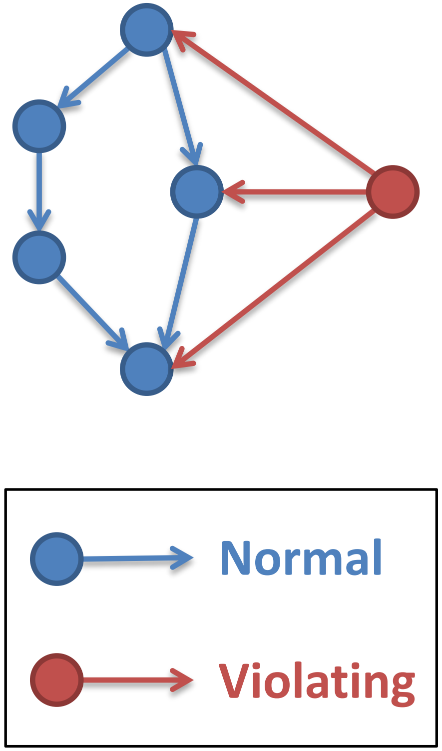
  – Example: buffer overflow

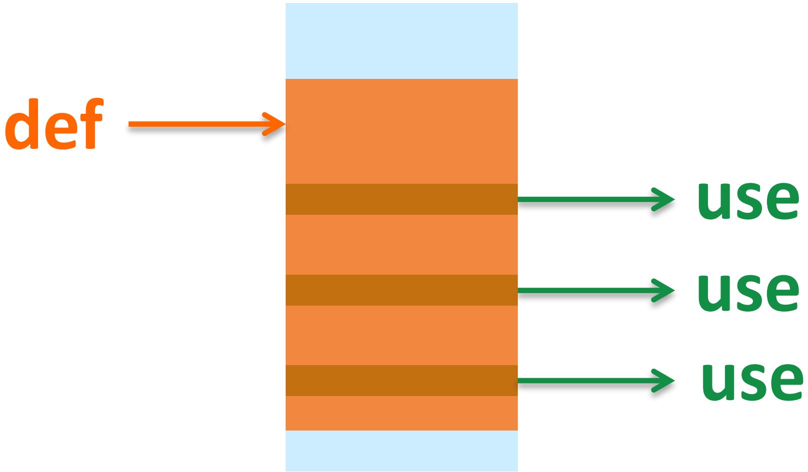




Data-flow Analysis Heuristics

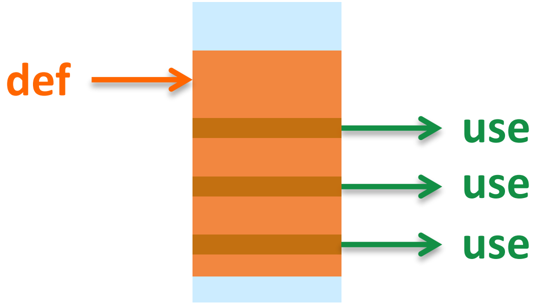
- One **def**, many **uses**:
  def is closer to the vulnerability
  - Example: buffer overflow

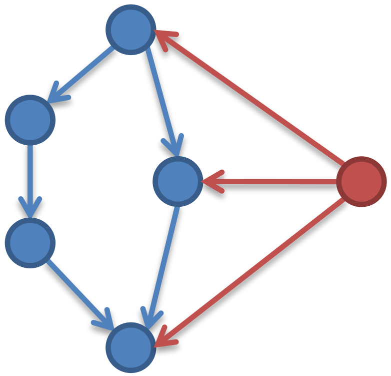




Data-flow Analysis Heuristics

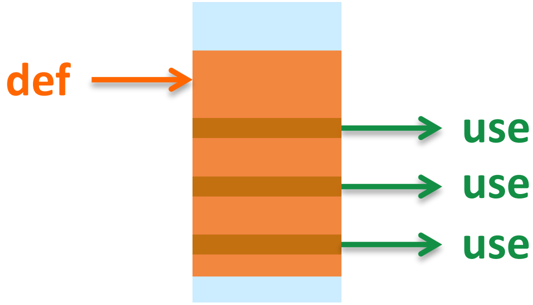
- One **def**, many **use**s:
  def is closer to the vulnerability
  - Example: buffer overflow

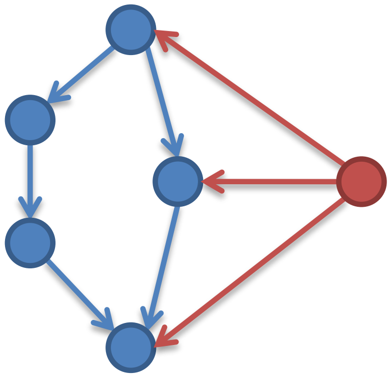




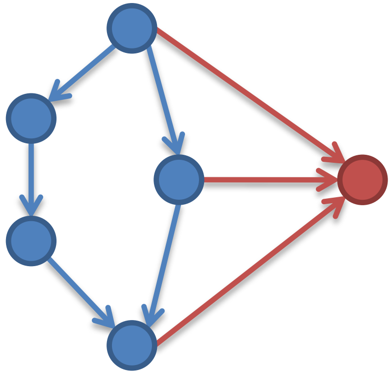
Data-flow Analysis Heuristics

- One **def**, many **use**s:
  def is closer to the vulnerability
  - Example: buffer overflow





- Many **def**s, one **use**:
  use is closer to the vulnerability
  - Example: information leakage



- …

Integer Errors

- Focus on common integer errors
  - Start from common functions/instructions that take integer operands
    - E.g., memcpy, recvfrom; movs, stos…
  - Search backwards for integer errors

- Example:

  memcpy (void * destination, const void * source, size_t num);

  Search from num backwards for integer errors.

Integer Errors

- **Assignment truncation** (e.g., 0x12345678 → 0x5678)
  - **To detect**: assign from a longer to a shorter integer type
- **Integer overflow/underflow** (e.g., 0xFFFFFFFF + 1)
  - **To detect**: check the RFLAGS register
- **Signedness error** (e.g., unsigned_int_var = signed_int_var)
  - **To detect**: collect hints from functions and instructions
    - Instructions: jg, jge, ja, jae, cmovg, cmova, idiv, div, etc.
    - Functions: memmove, strncat, etc.

Integer Errors

- **Assignment truncation** (e.g., 0x12345678 → 0x5678)
  - **To detect**: assign from a longer to a shorter integer type
- **Integer overflow/underflow** (e.g., 0xFFFFFFFF + 1)
  - **To detect**: check the RFLAGS register
- **Signedness error** (e.g., unsigned_int_var = signed_int_var)
  - **To detect**: collect hints from functions and instructions
    - Instructions: jg, jge, ja, jae, cmovg, cmova, idiv, div, etc.
    - Functions: memmove, strncat, etc.

- *Benign* integer errors?
  - Related to a reported vulnerability!

Use-after-free and Double-free

- Ravel instruments memory allocation/free functions to track the memory **life-time**
- **Use-after-free**: freed memory is accessed again
- **Double-free**: memory freed more than once without re-allocation

Race Condition

- When race condition happens, the execution deviates from the recorded one
  - as we do not implement strict R&R
- When detected, use the **happens-before** relation to check for race conditions

Implementation

- Record & replay:
  - FreeBSD release 10.2
  - Kernel modification + small user-space utility
- Vulnerability locator:
  - Extended from Valgrind

Evaluation – Effectiveness

- Buffer overflow
- Integer errors
- Information leakage
- Use-after-free and double-free
- Format string vulnerabilities

CVE-2013-2028 of Nginx

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
...
size = (size_t) ngx_min(r->headers_in.
     content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
     );
n=r->connection->recv(r->connection, buffer, size);
```

CVE-2013-2028 of Nginx

signed comparison

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
...
size = (size_t) ngx_min(r->headers_in.
     content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
     );
n=r->connection->recv(r->connection, buffer, size);
```

unsigned signed

CVE-2013-2028 of Nginx

signed comparison

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
...
size = (size_t) ngx_min(r->headers_in.
     content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
     );
n=r->connection->recv(r->connection, buffer, size);
```

unsigned

signed

larger than expected

CVE-2013-2028 of Nginx

signed comparison

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
...
size = (size_t) ngx_min(r->headers_in.
    content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
    );
n=r->connection->recv(r->connection, buffer, size);
```

unsigned

signed

larger than expected

buffer overflow

CVE-2013-2028 of Nginx

Ravel

signed comparison

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
...
size = (size_t) ngx_min(r->headers_in.
     content_length_n, NGX_HTTP_DISCARD_BUFFER_SIZE
     );
n=r->connection->recv(r->connection, buffer, size);
```

unsigned

signed

larger than expected

buffer overflow

Signedness Conflict

Data-flow Violation

Memory Exception

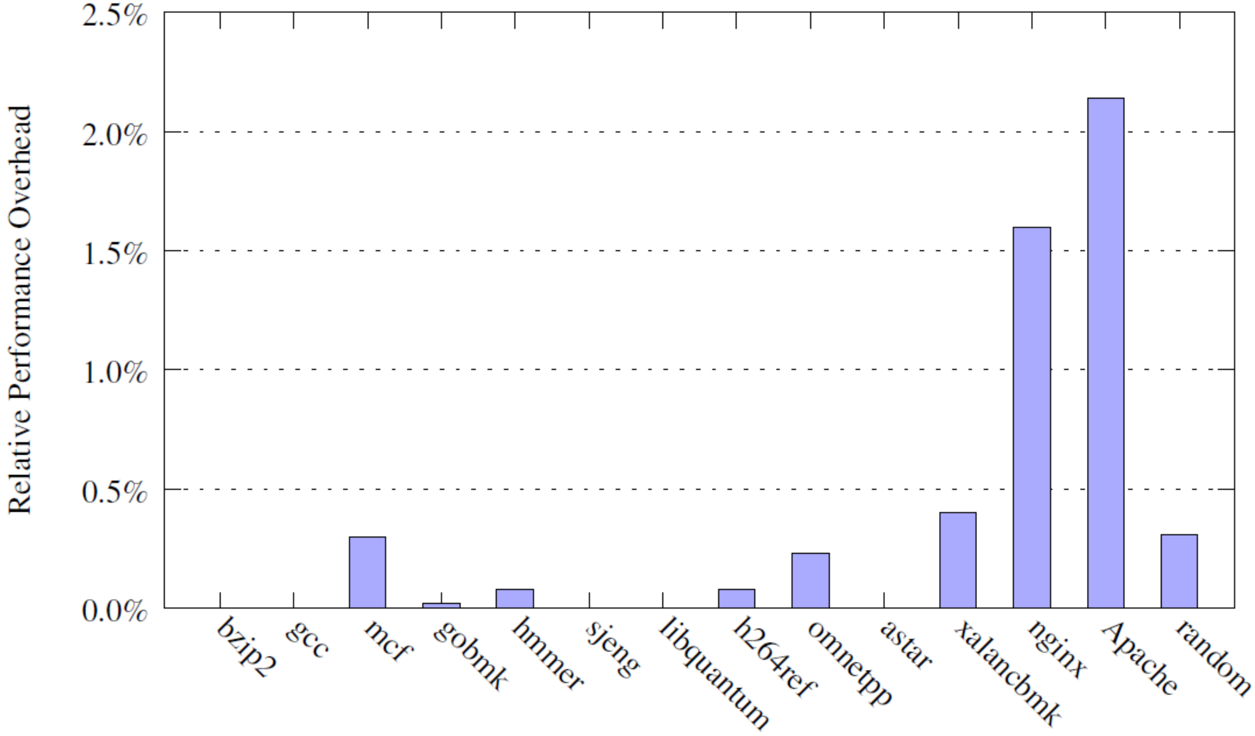
Evaluation – Effectiveness

- More examples are in the paper (Heartbleed, etc.)

Program Name	Vulnerability Type	Pinpointed?
BitBlaster	Null Pointer Derefernece	Yes
CGC_Planet_Markup_Language_Parser	Heap Overflow	Yes
	NULL Pointer Dereference	Yes
	StackOverflow	Yes
CGC_Board	Heap Overflow	Yes
CGC_Symbol_Viewer_CSV	Integer Overflow	Yes
CGC_Video_Format_Parser_and_Viewer	Heap Overflow	Yes
simple_integer_calculator	Heap Overflow	Yes
	Integer Overflow	Yes
	NULL Pointer Dereference	Yes
	Out-of-bounds Read	Yes
Diary_Parser	Null Pointer Dereference	Yes
	Out-of-bounds Read	Yes
	Stack Overflow	Yes
electronictrading	Heap Overflow	Yes
	Integer Overflow	Yes
	Untrusted Pointer Dereference	Yes
	Use After Free	Yes
Enslavednode_chat	Heap Overflow	Yes
Kaprica_Script_Interpreter	Arbitrary Format String	Yes
	NULL Pointer Dereference	Yes
KTY_Pretty_Printer	Double Free	Yes
	Stack Overflow	Yes

Table 1: Summary of the evaluation results on a number of DARPA CGC programs.

Evaluation – Performance



Performance overhead of Ravel's online components relative to the original FreeBSD system

Pinpointing Vulnerabilities

Q&A

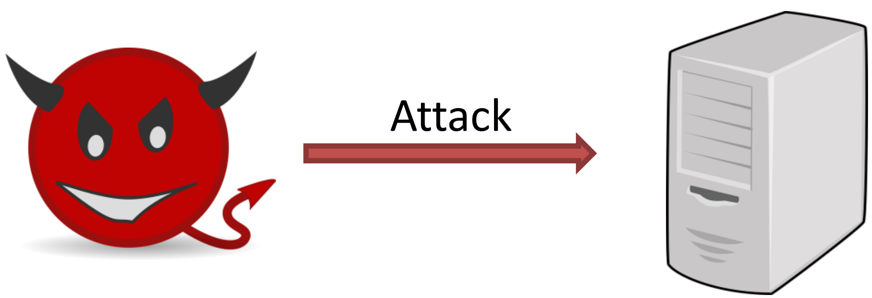
http://YueChen.me

Backup Slides

Attack Detection Example

- Typical scenario example:

Attacker guesses memory addresses → **Program crashes** (due to ASLR, DEP, etc.) → **Victim forks a new process**



Attack Detection Example

- Typical scenario example:

Attacker guesses memory addresses → **Program crashes** (due to ASLR, DEP, etc.) → **Victim forks a new process**