CS 578: CYBER-SECURITY

PART IV: MALWARE

Sanghyun Hong
sanghyun.hong@oregonstate.edu





RECAP – WHERE WE ARE

- Part I – Network/Internet security
  - The attacker is outside your machine
- Part II – Computer systems security
  - The attacker is inside your machine
  - Memory safety: how the attacker gets in
  - OS security: decide what the attacker may do (access control)
- Part III – Isolation and (software-induced) breaks
  - The attacker is inside your program or is the OS itself
  - The attacker exploits your hardware and breaks integrity and confidentiality
- Part IV – Software/Web security
  - The attacker turns bugs in your software into exploits (today!)

SOFTWARE VULNERABILITIES – WHY BUGS MATTER

HUMANS MAKE ERRORS

- Suppose that we manufacture products
- We make errors if
  - We are under stress
  - We worked too many hours
  - We face a quick production cycle (e.g., one day)
  - … (many more)

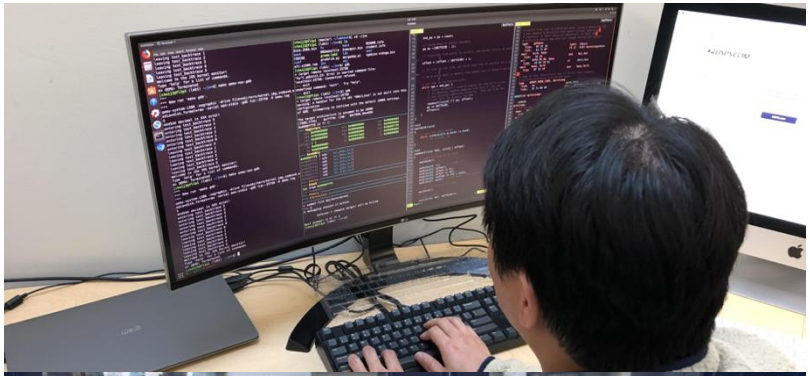






HUMANS MAKE ERRORS

- We develop software
  - Humans are prone to making errors
  - Humans make more mistakes if
    - They are too stressed from work
    - They are too stressed from life
    - Work is hard
    - Worked too many hours (60+ hrs/wk)
    - A quick development cycle (sprints)
    - … (many more)





MODERN SOFTWARE IS COMPLEX

- Google Chrome
  - 4.5M lines of code 10 years ago



Shashwat Anand

Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

Used Cloc [http://cloc.sourceforge.net/] just like Dan Loewenherz did for the question How many lines of code are in the Linux kernel?

MODERN SOFTWARE IS COMPLEX

- Google Chrome
  - 4.5M lines of code 10 years ago



Shashwat Anand

Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

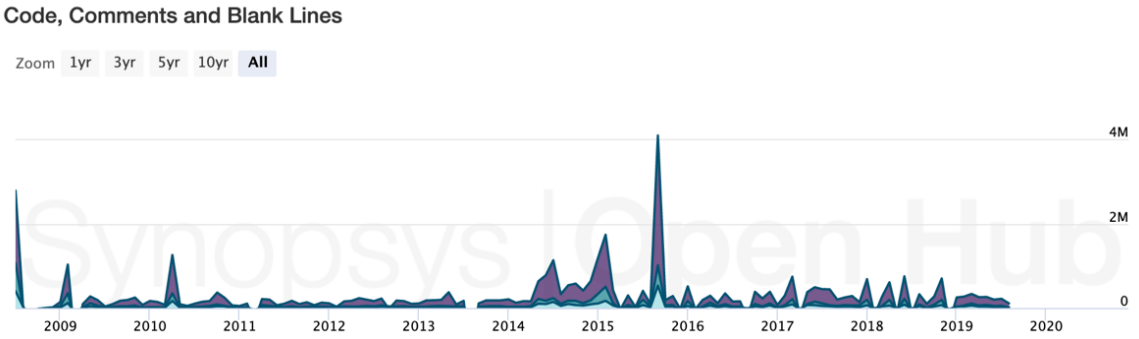
4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

Used Cloc [http://cloc.sourcefo

How many lines of code are in th

Total Lines :	34,900,821	Code Lines :	25,683,389	Percent Code Lines :	73.6%
Number of Languages :	36	Total Comment Lines :	4,603,400	Percent Comment Lines :	13.2%
		Total Blank Lines :	4,614,032	Percent Blank Lines :	13.2%

  - >34M lines today



MODERN SOFTWARE IS COMPLEX

- Linux kernel
  - >12M lines of code in 2015
  - >27M lines of code in 2020

- Android
  - Android 1.6: >4.5M lines in 2009
  - Android 5.1: > 9M lines in 2014
  - Android 8.0: > 25M lines in 2017

- Humans are prone to making errors
- Work environment often makes people more prone to making errors in code
- The complexity in software makes it more difficult for humans to follow the code (Complexity: $O(N^2)$ where N = lines of code)
- …

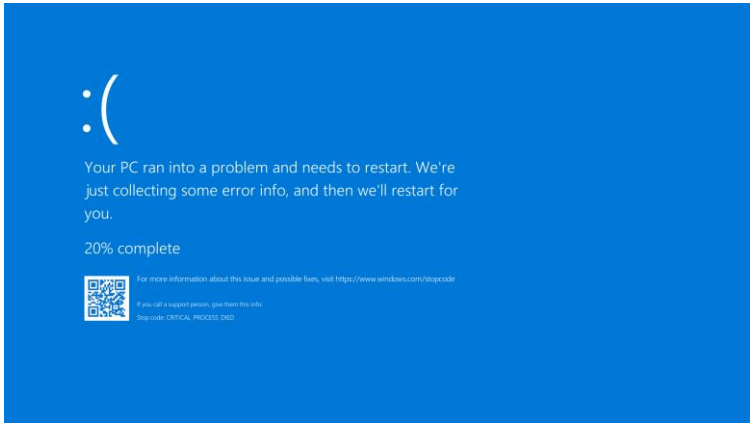
WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash

```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x-evm-07.00.00.00-Linux-x86-Install.bin
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$
```





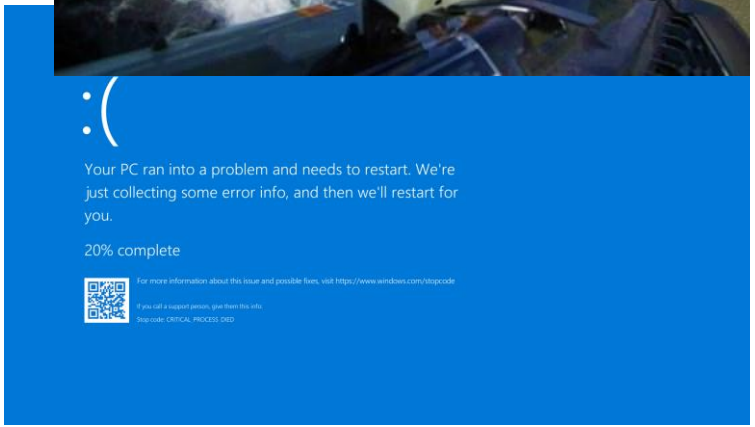
WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash



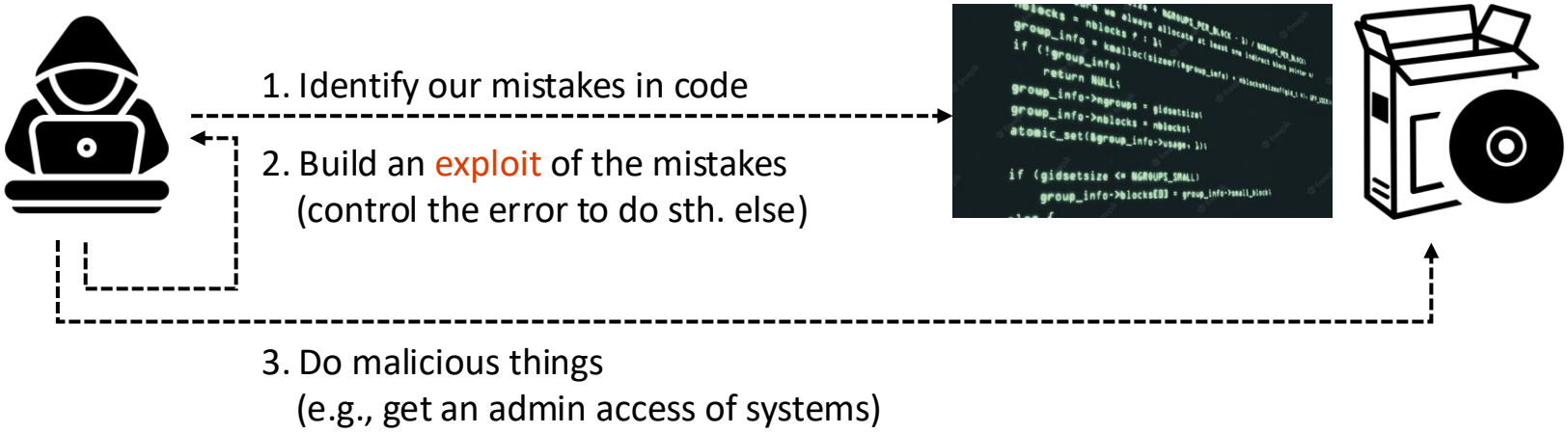
```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x-
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$
```





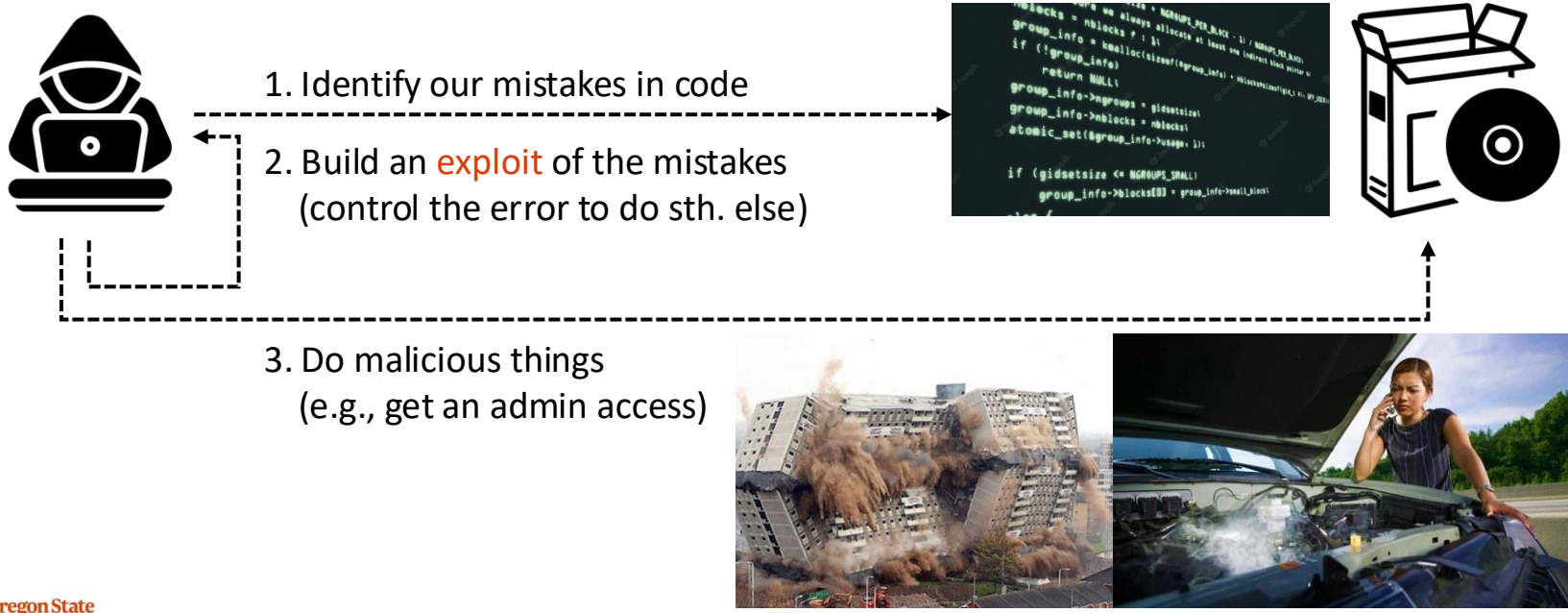
WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash
- A hack



WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash
- A hack



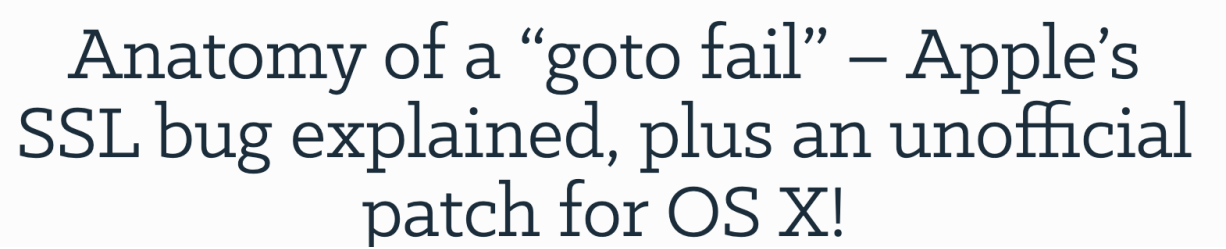
VULNERABILITY AND EXPLOIT

- Vulnerability
  - An error (= a bug) that can be exploited by an adversary
  - The attacker can make the software do what it was not meant to do
  - Public lists: CVE (individual bugs), CWE (types of bugs)

- Exploit
  - An input that triggers a vulnerability with malicious intent
  - A proof-of-concept program showing how an attacker could use the bug
  - You can find sample exploits from, e.g., Metasploit

MOTIVATING EXAMPLE: GOTO FAIL

- In 2014



MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- In 2014



About the security content of iOS 7.0.6

This document describes the security content of iOS 7.0.6.

iOS 7.0.6

- **Data Security**

Available for: iPhone 4 and later, iPod touch (5th generation), iPad 2 and later

Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS

Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

CVE-ID

CVE-2014-1266

Why???
What was the mistake??

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code above the red square is okay
  - They run SHA1 and check errors
- The code in the red boxes:
  - It does not fall into any if statement
  - It always leads to "goto fail;"
  - err is still 0 (= success)
  - sslRawVerify() never runs

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- How could an attacker use this?
  - Suppose an attacker runs public Wi-Fi
  - ex) 'PDX Free WiFi', 'eduroam', …
  - You open https://your-bank.com
  - The attacker answers instead
  - Real certificate, fake signature
  - Your phone skips the signature check
  - Now the attacker can read and modify everything you send

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL

- Small mistake; big impact
  - A mistake: adds one additional line of ‘goto fail’
  - Result: attackers may hijack a TLS-protected connection
  - Impact: attackers may read/modify TLS connections from iOS and OS X

- Implications
  - Even a simple mistake could lead to a disaster
  - Errors are not random like a natural disaster
  - Errors can be controlled (‘exploited’) by attackers

PROGRAM ANALYSIS – FINDING VULNERABILITIES

PROGRAM ANALYSIS

- Static analysis
  - Examine the source code without running it
  - Search for the patterns that lead to security vulnerabilities – signatures
  - Use these signatures on the source code-level

- Dynamic analysis
  - Run the program and observe its behavior
  - Search for the (runtime) patterns that lead to security vulnerabilities – signatures
  - Catch vulnerabilities that only show up at runtime

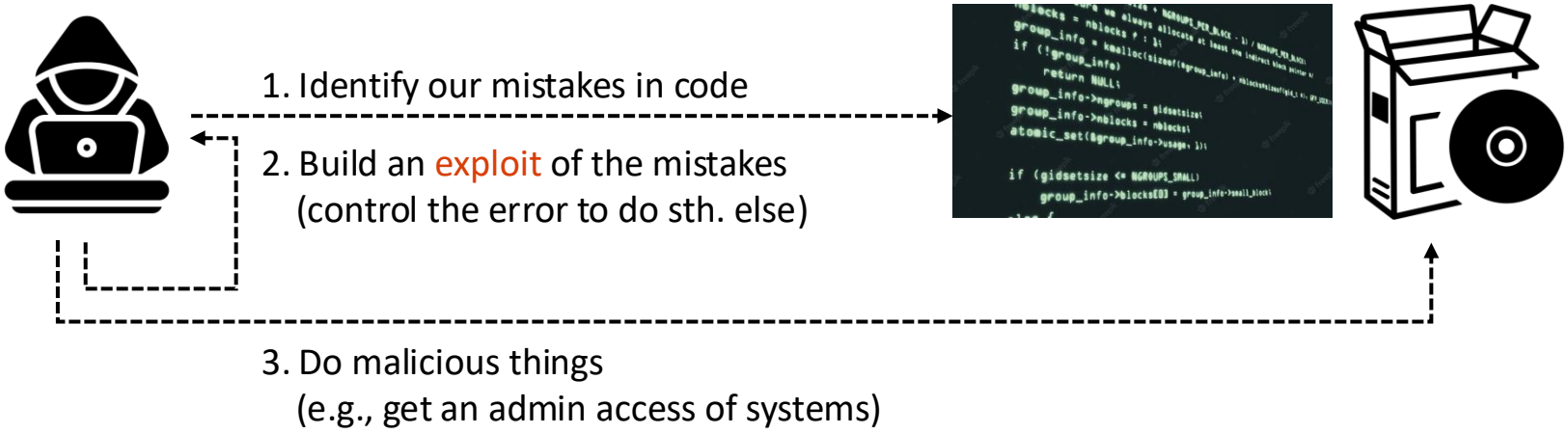
PROGRAM ANALYSIS

- Static analysis
  - Examine the source code without running it
  - Search for the patterns that lead to security vulnerabilities – signatures
  - Use these signatures on the source code-level

- Implications
  - Around 2010, finding vulnerabilities was mostly manual
  - There could be unresolved or unknown vulnerabilities in the source code
  - Those vulnerabilities could be used for 0-day attacks (or miscreants sell them)

VULNERABILITY != EXPLOITATION

- Crash
- A hack



VULNERABILITY != EXPLOITATION – CONT'D

- Motivation
  - Given a program
    - Automatically find vulnerabilities
    - Generate exploits for them



AEG – AUTOMATIC EXPLOIT GENERATION

AEG – WHY IS IT HARD?

- Goal
  - Tell which bugs are exploitable
- ex) Copy 12 bytes into a 10-byte buffer
  - A bug in the source code

```
char src[12], dst[10];
strncpy(dst, src, sizeof(src));
```

In the source code



dst: 10 bytes

strncpy copies 12 bytes



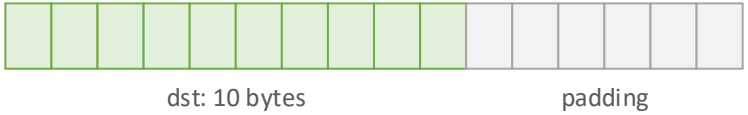
2 bytes past dst

AEG – WHY IS IT HARD? – CONT'D

- Goal
  - Tell which bugs are exploitable
- ex) Copy 12 bytes into a 10-byte buffer
  - A bug in the source code
- After compilation
  - The compiler pads dst to 16 bytes

```
char src[12], dst[10];
strncpy(dst, src, sizeof(src));
```

In the binary



AEG – WHY IS IT HARD? – CONT'D

- Goal
  - Tell which bugs are exploitable
- ex) Copy 12 bytes into a 10-byte buffer
  - A bug in the source code
- After compilation
  - The compiler pads dst to 16 bytes
  - The copy stays inside dst
  - A bug, but not exploitable

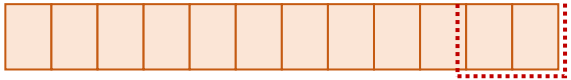
```
char src[12], dst[10];
strncpy(dst, src, sizeof(src));
```

In the binary



dst: 10 bytes padding

strncpy copies 12 bytes



The extra 2 bytes land in the padding.
Nothing else changes.

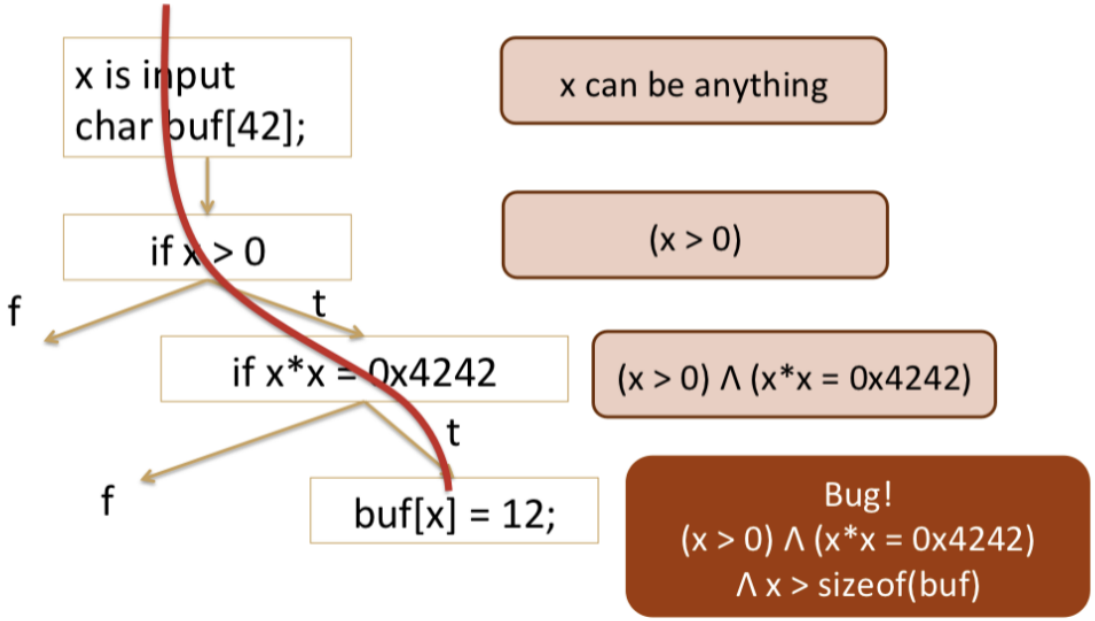
AEG – WHY IS IT HARD? – CONT'D

- Source code alone is not enough
  - Exploitable or not depends on the binary
- Binary alone does not scale
  - No variables, types, or functions
- AEG uses both
  - Find bugs in the source code
  - Check them on the binary



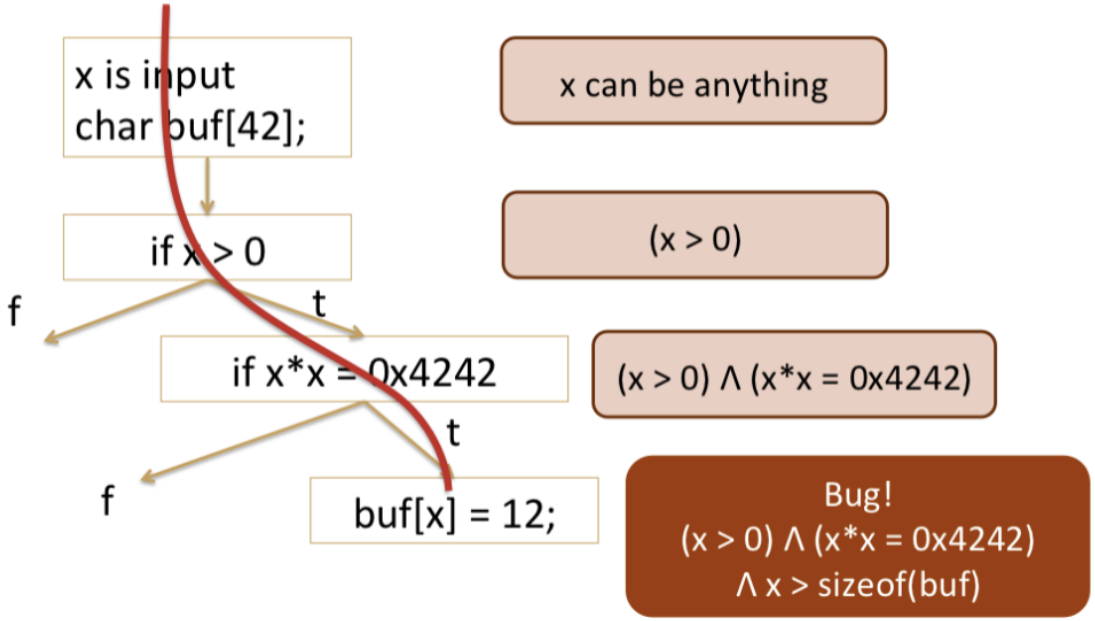
PRELIM – SYMBOLIC EXECUTION

- Symbolic execution
  - Input x is a symbol
  - Follow both branches
  - Keep the conditions



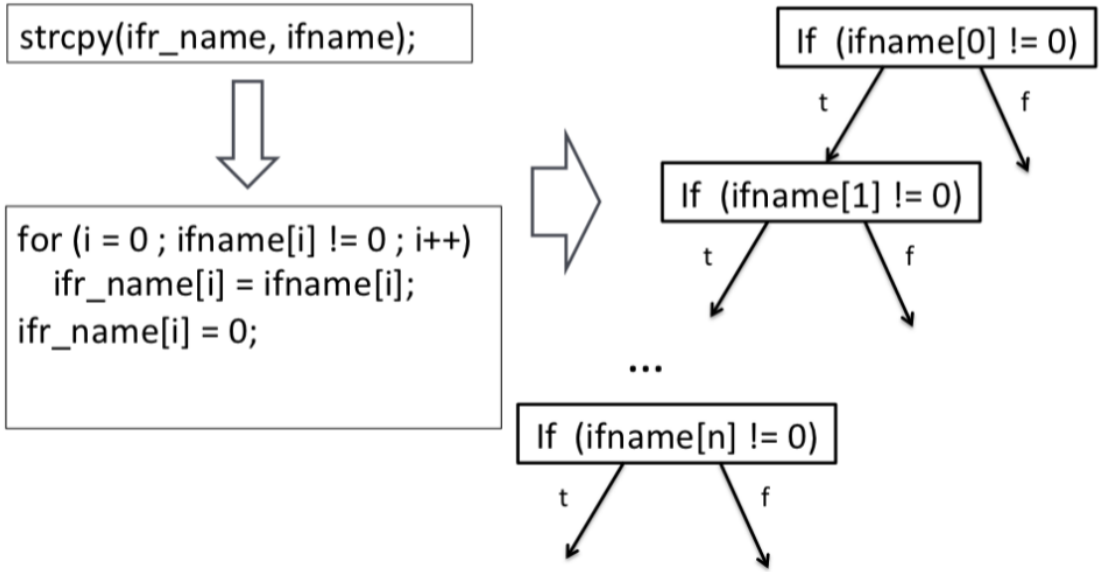
PRELIM – SYMBOLIC EXECUTION

- Symbolic execution
  - Input x is a symbol
  - Follow both branches
  - Keep the conditions
- Path constraint
  - Conditions on one path
  - Solver finds such an x
  - That x hits the bug



PRELIM – PATH EXPLOSION

- Path explosion
  - strcpy is a loop
  - One branch per byte
  - n bytes: n + 1 paths
  - Nested loops multiply
  - Too many to check all



AEG – OVERVIEW

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)

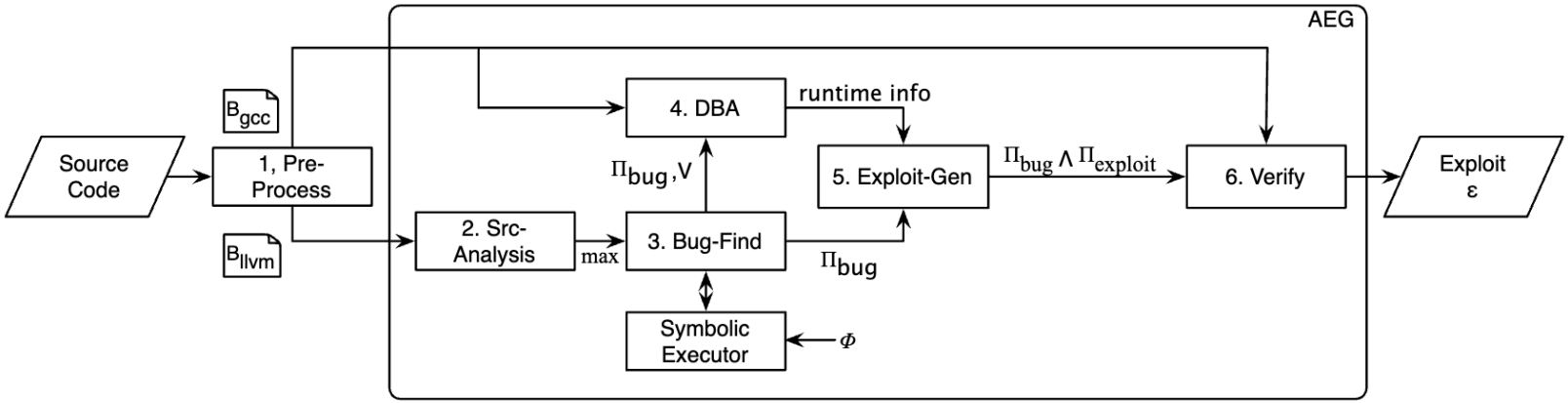
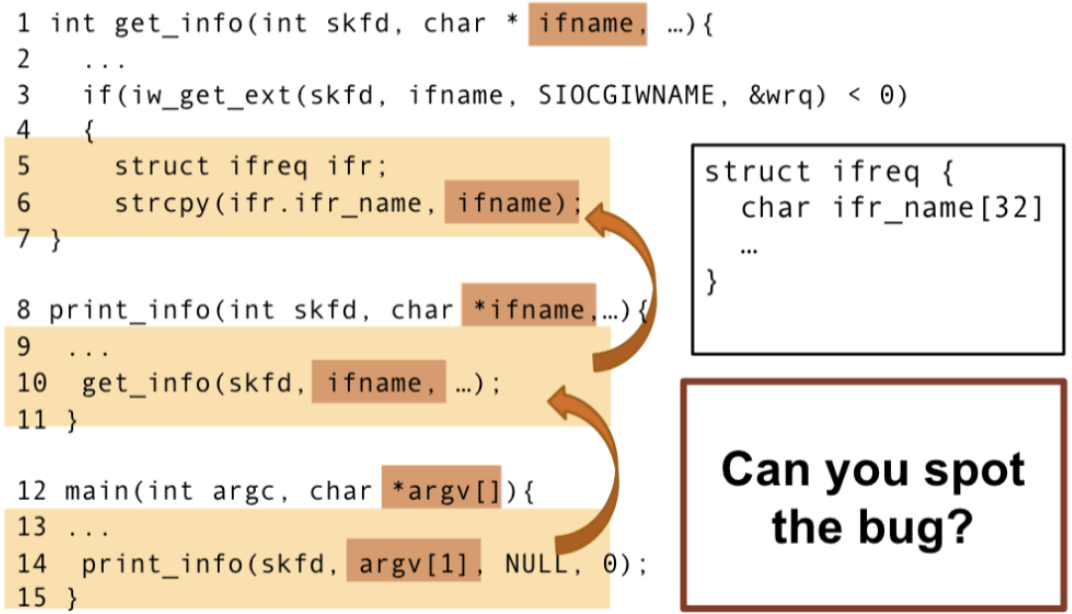


Figure 5: AEG design.

Π: path constraint ε: exploit

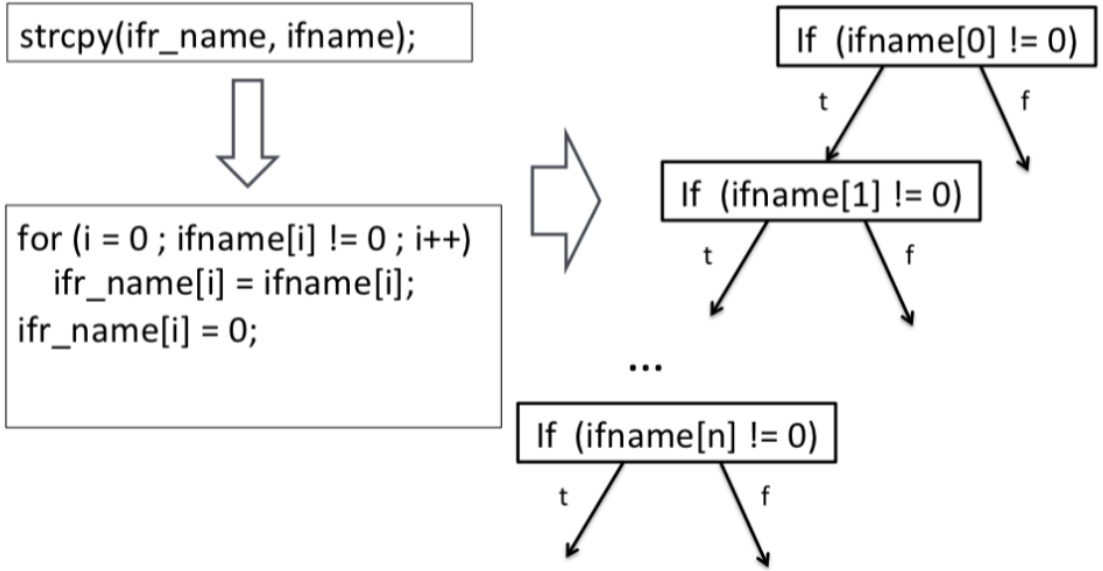
AEG – BUG-FIND

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - **Bug hunt**
    - ex. iwconfig



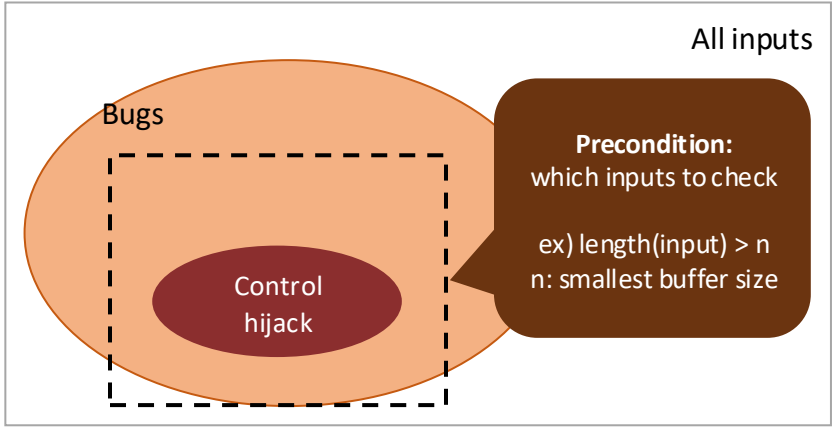
AEG – BUG-FIND

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - **Bug hunt**
    - ex. iwconfig
    - Symbolic execution
    - Prior work: KLEE
      - Scalability issue
      - Need to prove the absence of bugs by running all the paths



AEG – PRECONDITIONED SYMBOLIC EXECUTION

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - **Bug hunt**
    - Prior work: KLEE
      - Scalability issue
      - Need to prove the absence of bugs by running all the paths
    - AEG: preconditioned symbolic execution



AEG – PRECONDITIONED SYMBOLIC EXECUTION

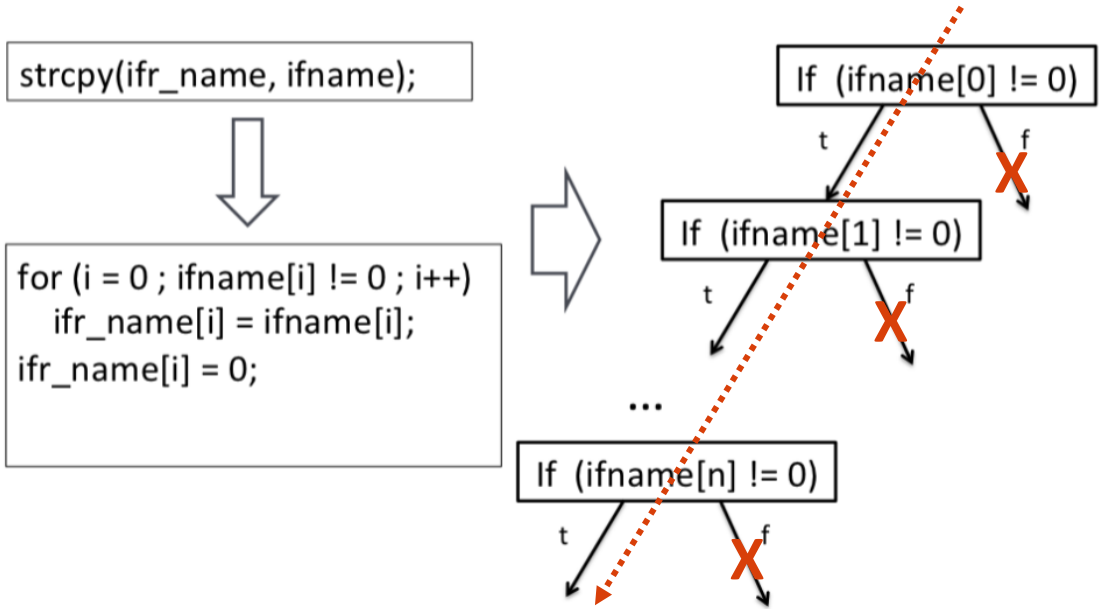
- Precondition: explore only inputs that satisfy it

Precondition	What is fixed	Example
None	Nothing	Plain symbolic execution
Known length	The input length	Input 10% longer than the largest buffer
Known prefix	The first bytes	An email address must have “@” (xmail)
Concolic	Every byte	One full input, e.g., an HTTP GET request

More fixed: faster, but may miss exploits
Less fixed: closer to exploring every path

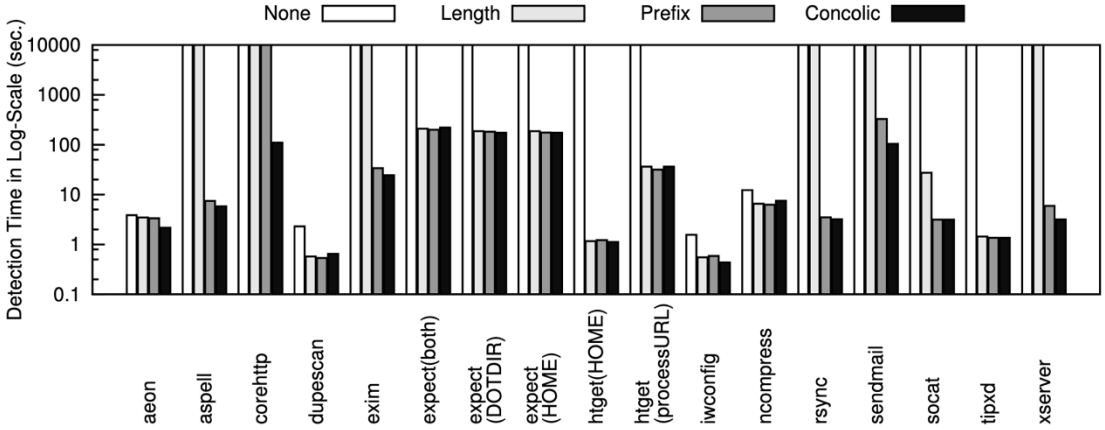
AEG – PRECONDITIONED SYMBOLIC EXECUTION

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - **Bug hunt**
    - ex. iwconfig
    - Symbolic execution
    - Prior work: KLEE
    - AEG: preconditioned symbolic execution + path prioritization



AEG – DOES THE PRECONDITION HELP?

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - **Bug hunt**
    - ex. iwconfig
    - Symbolic execution
    - Prior work: KLEE
    - AEG: preconditioned symbolic execution + path prioritization



Time to find the bug per program, for each precondition
(log scale; a full bar = not found within 10,000 s)

AEG – WHICH PATHS FIRST?

- Buggy-path-first
  - A small mistake on a path hints at more bugs further along
  - ex) tipxd: malloc(strlen(s)) is one byte short (not exploitable)
  - The same path later has an exploitable format string bug
- Loop exhaustion
  - Run loops over symbolic input to the end first
  - Overflows (e.g., strcpy) happen at the end of the loop

AEG – DYNAMIC BINARY ANALYSIS

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - **DBA: Dynamic binary analysis**

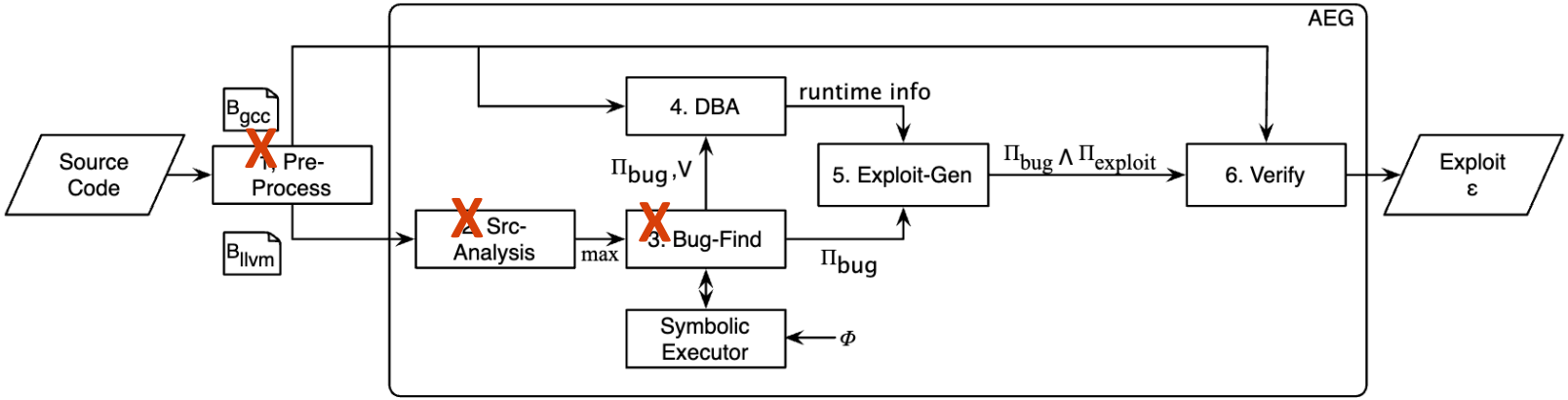


Figure 5: AEG design.

AEG – DYNAMIC BINARY ANALYSIS

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - **DBA**
    - Information from the source code is insufficient for generating an exploit
    - Runtime information is needed
      - Think about your buffer overflow (HW 2)
      - ex) Where %rip is saved when a function is called

AEG – DYNAMIC BINARY ANALYSIS

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - **DBA**
    - Information from the source code is insufficient for generating an exploit
    - Runtime information is needed
    - Process
      - Take the concrete input for the bug from “Bug-Find”
      - Set breakpoints on the vulnerable function (also found by “Bug-Find”)
      - Collect the runtime information the exploit needs

AEG – EXPLOIT GENERATION

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - **Exploit-Gen**

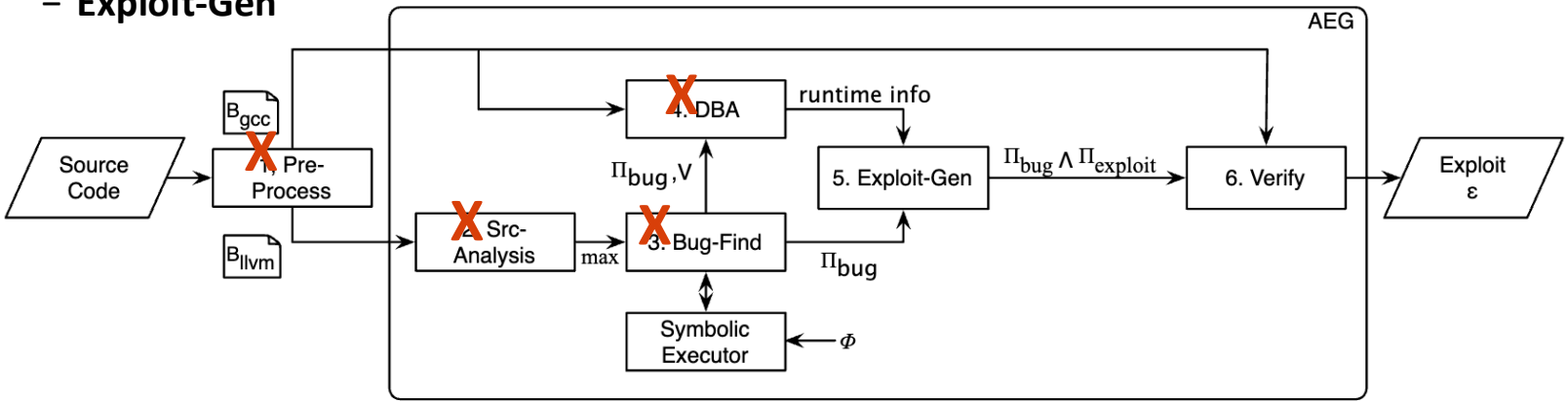


Figure 5: AEG design.

AEG – EXPLOIT GENERATION

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - **Exploit-Gen**
    - Takes
      - Path constraints
      - Runtime information, e.g., vulnerable variable (buffer) address
      - Runtime stack memory constraints
    - Generates
      - Exploit formulas
        - ≫ Stack overflow (return to libc and return to stack)
        - ≫ Format string (return to libc and return to stack)

AEG – VERIFY

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - Exploit-Gen
  - **Verify**

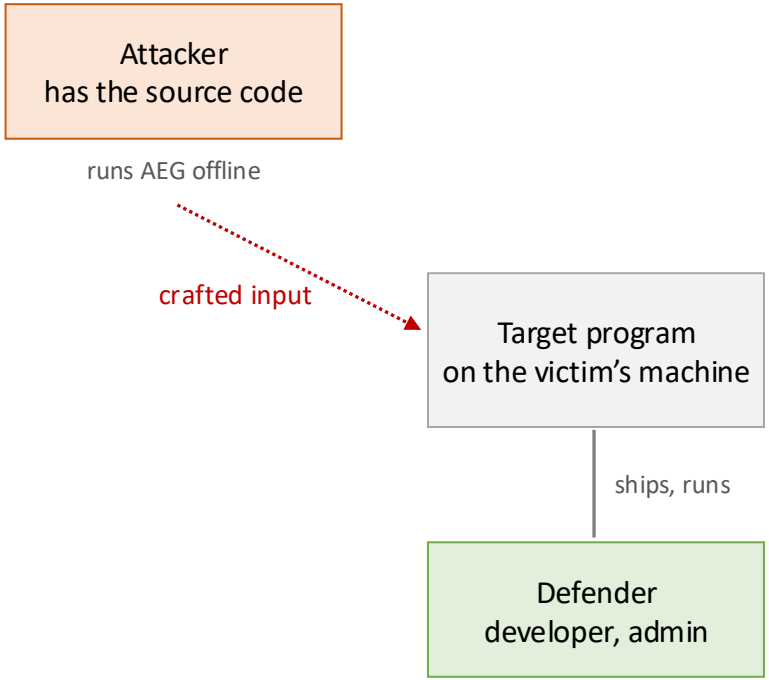
	Program	Ver.	Exploit Type	Vulnerable Input src	Gen. Time (sec.)	Executable Lines of Code	Advisory ID.
None	aeon	0.2a	Local Stack	Env. Var.	3.8	3392	CVE-2005-1019
	iwconfig	V.26	Local Stack	Arguments	1.5	11314	CVE-2003-0947
	glftpd	1.24	Local Stack	Arguments	2.3	6893	OSVDB-ID#16373
	ncompress	4.2.4	Local Stack	Arguments	12.3	3198	CVE-2001-1413
Length	htget (processURL)	0.93	Local Stack	Arguments	57.2	3832	CVE-2004-0852
	htget (HOME)	0.93	Local Stack	Env. Var	1.2	3832	Zero-day
	expect (DOTDIR)	5.43	Local Stack	Env. Var	187.6	458404	Zero-day
	expect (HOME)	5.43	Local Stack	Env. Var	186.7	458404	OSVDB-ID#60979
	socat	1.4	Local Format	Arguments	3.2	35799	CVE-2004-1484
	tipxd	1.1.1	Local Format	Arguments	1.5	7244	OSVDB-ID#12346
Prefix	aspell	0.50.5	Local Stack	Local File	15.2	550	CVE-2004-0548
	exim	4.41	Local Stack	Arguments	33.8	241856	EDB-ID#796
	xserver	0.1a	Remote Stack	Sockets	31.9	1077	CVE-2007-3957
	rsync	2.5.7	Local Stack	Env. Var	19.7	67744	CVE-2004-2093
	xmail	1.21	Local Stack	Local File	1276.0	1766	CVE-2005-2943
Concolic	corehttp	0.5.3	Remote Stack	Sockets	83.6	4873	CVE-2007-4060
Average Generation Time & Executable Lines of Code					114.6	56784	

AEG – RESULTS

- Setup
  - 14 open-source programs from CVE, OSVDB, Exploit-DB
- Results
  - 16 working exploits, each gives a shell
  - 2 zero-days: expect 5.43 and htget 0.93
  - 114.6 seconds on average; the slowest (xmail) took 1,276 s
  - 10 of 16 needed no help from the user (none or length)
  - KLEE alone found only the iwconfig bug

AEG – ATTACKER AND DEFENDER

- Attacker
  - Wants a shell on the victim's machine
  - Has the source code of a program it runs
  - Runs AEG offline, then sends the input
- Defender
  - Developer who ships the program
  - Admin who runs it
- AEG works for both sides
  - Defender runs it first
  - Fix the exploitable bugs first



AEG – THREAT MODEL

- Scenario
  - The attacker has the source code of a program
  - The program reads input from args, env. vars, files, or sockets
- Knowledge
  - The source code and the binary built for the target
- Capabilities
  - Run and debug the binary on their own machine
  - Use a symbolic executor and a constraint solver
- Assumptions
  - No ASLR and no W⊕X (DEP); stack overflows and format strings only

AEG – COUNTERMEASURES

- What the defender can do

Attacker needs	Defender's countermeasure	Problem
Source code	Do not release source	Decompilers (used at a CTF)
Fixed memory layout	ASLR	Address leaks
Executable stack	W⊕X (DEP)	Reusing existing code
Unchecked copies	Bounds checks, safer APIs	Old code; size bugs remain
Unknown bugs	Defenders run AEG first	Path explosion

CONTINUOUS LEARNING FOR ANDROID MALWARE DETECTION IN-CLASS PRESENTATION BY YUFENG

RECAP – WHERE WE ARE

- Part I – Network/Internet security
  - The attacker is outside your machine
- Part II – Computer systems security
  - The attacker is inside your machine
  - Memory safety: how the attacker gets in
  - OS security: decide what the attacker may do (access control)
- Part III – Isolation and (software-induced) breaks
  - The attacker is inside your program or is the OS itself
  - The attacker exploits your hardware and breaks integrity and confidentiality
- Part IV – Software/Web security
  - The attacker turns bugs in your software into exploits

Thank You!

Sanghyun Hong

https://true-lab.ai/courses/Sec-Grad/current



