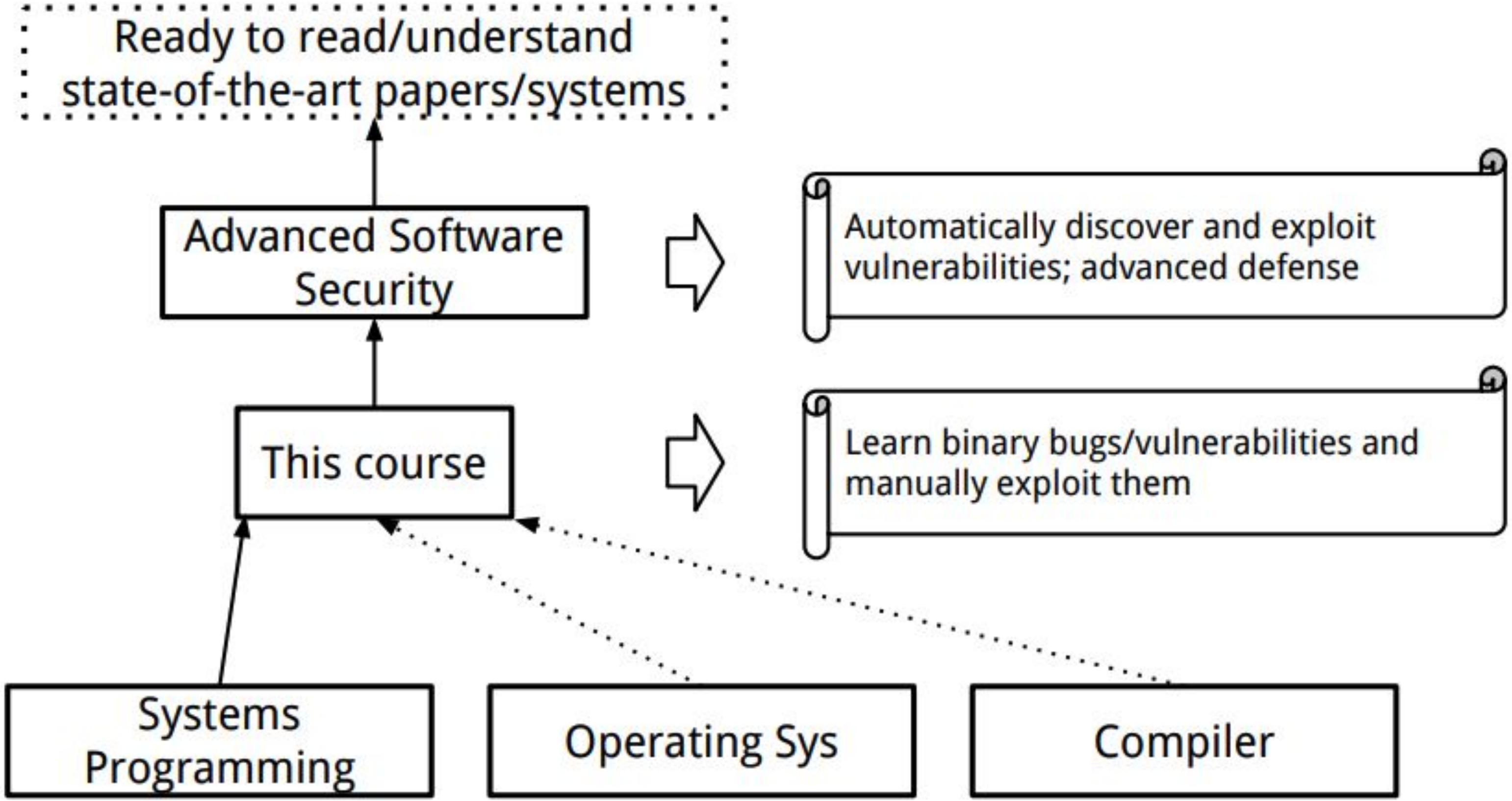
System Security - Attack and Defense for Binaries

CS 4390/5390, Spring 2026

Instructor: MD Armanuzzaman (***Arman***)

If you want to be a systems/software security guy …



What we have learned so far?

- Vulnerabilities
  - Buffer overflow (out of bound read/write), format string vulnerability, heap vulnerabilities...
- Exploitation techniques
  - Return to Libc
  - Shellcode development
  - Return-oriented programming
  - ...
- Mitigations at hardware, OS, compiler layers
  - Non-execution stack
  - Control-flow integrity
  - ASLR
  - ...

The workflow of exploitation

1. Discover and understand the vulnerabilities or bugs
    a. code review, reverse engineering
    b. root cause, what memory is affected, memory address calculation
2. Assess exploitability
    a. what do we control? what mitigations are active?
3. Choose technique and develop exploit
    a. based on mitigations and available primitives
    b. craft payload, bypass protections, gain control

Manually discover real-world bugs is not alway easy

```
static void
xmlSnprintfElementContent(char *buf, int size,
                          xmlElementContentPtr content, int englob)
{
    int len = strlen(buf);   // (1) how much is in the buffer

    if (len > size - 10) return;    // (2) safety check
    switch (content->type) {
        case XML_ELEMENT_CONTENT_PCDATA:
            strcat(buf, "#PCDATA");  // (3) append "#PCDATA"
            break;
        case XML_ELEMENT_CONTENT_ELEMENT:
            // ... append element name ...
            strcat(buf, content->name);
            break;

READ THE NEXT CODE BLOCK ⇒
```

```
        case XML_ELEMENT_CONTENT_SEQ:
        case XML_ELEMENT_CONTENT_OR:
            if (englob) strcat(buf, "(");          // (4) append "("
            xmlSnprintfElementContent(buf, size,
                                content->c1, 0);  // (5) recurse
            if (content->type == XML_ELEMENT_CONTENT_SEQ)
                strcat(buf, " , ");               // (6) append
            else
                strcat(buf, " | ");
            xmlSnprintfElementContent(buf, size,
                                content->c2, 0);  // (7) recurse again
            if (englob) strcat(buf, ")");          // (8) append ")"
            break;
    }
}
```

CVE-2017-9047 — libxml2, xmlSnprintfElementContent() in valid.c

A widely-used C library for parsing XML. Used by almost every Linux application, browser, and server that touches XML or HTML.

https://nvd.nist.gov/vuln/detail/cve-2017-9047

Manually discover real-world bugs is not alway easy

```
static void
xmlSnprintfElementContent(char *buf, int size,
                          xmlElementContentPtr content, int englob)
{
    int len = strlen(buf);   // (1) how much is in the buffer

    if (len > size - 10) return;    // (2) safety check
    switch (content->type) {
        case XML_ELEMENT_CONTENT_PCDATA:
            strcat(buf, "#PCDATA");  // (3) append "#PCDATA"
            break;
        case XML_ELEMENT_CONTENT_ELEMENT:
            // ... append element name ...
            strcat(buf, content->name);
            break;

READ THE NEXT CODE BLOCK ⇒
```

```
        case XML_ELEMENT_CONTENT_SEQ:
        case XML_ELEMENT_CONTENT_OR:
            if (englob) strcat(buf, "(");          // (4) append "("
            xmlSnprintfElementContent(buf, size,
                                  content->c1, 0);  // (5) recurse
            if (content->type == XML_ELEMENT_CONTENT_SEQ)
                strcat(buf, " , ");               // (6) append
            else
                strcat(buf, " | ");
            xmlSnprintfElementContent(buf, size,
                                  content->c2, 0);  // (7) recurse again
            if (englob) strcat(buf, ")");          // (8) append ")"
            break;
    }
}
```

Entry: len = size - 11 → check passes (size-11 < size-10) ✓

Step 5: recurse fills buffer to size - 4

Step 6: strcat(buf, " , ") → writes 3 bytes → buffer now at size - 1

Step 8: strcat(buf, ")") → writes 1 byte → OOB write ✗

Other reasons why manually discovering bugs is not feasible

Real-world codebases have millions of lines of code

- Linux kernel — ~40 million lines of code, growing by ~70,000 lines every release cycle
- Windows 10 — ~50 million lines of code
- macOS — over 80 million lines of code
- Google Chromium — ~20–30 million lines of code
- Android AOSP — ~7 million lines (core only, excluding external dependencies)

The workflow of exploitation

1. Discover and understand the vulnerabilities or bugs
    a.
    b. root cause, what memory is affected, memory address calculation

Automate bug discovery at scale

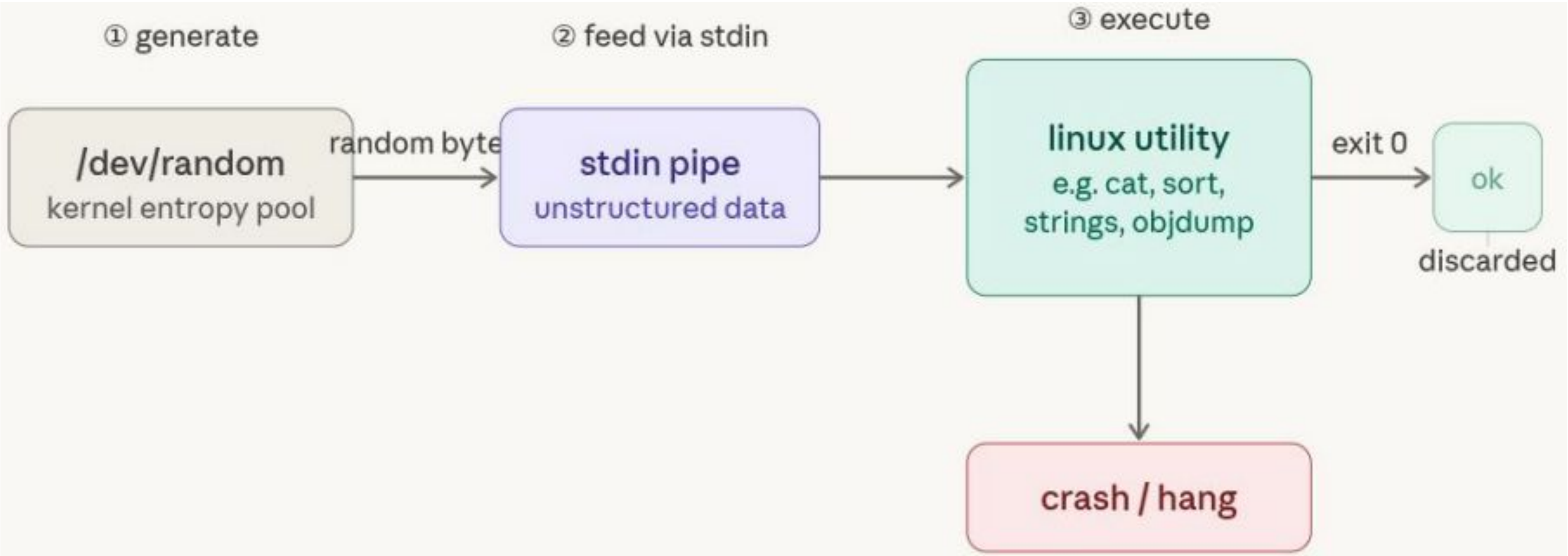
2. Assess exploitability
    a. what do we control? what mitigations are active?
3. Choose technique and develop exploit
    a. based on mitigations and available primitives
    b. craft payload, bypass protections, gain control

Fuzzing or Fuzz Testing

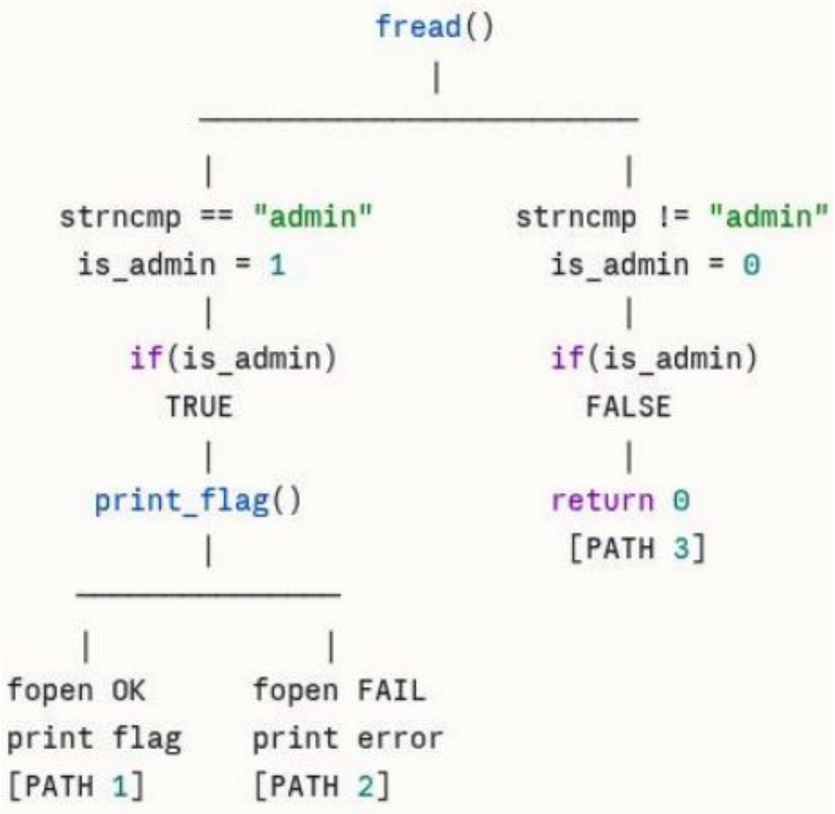
What is fuzzing

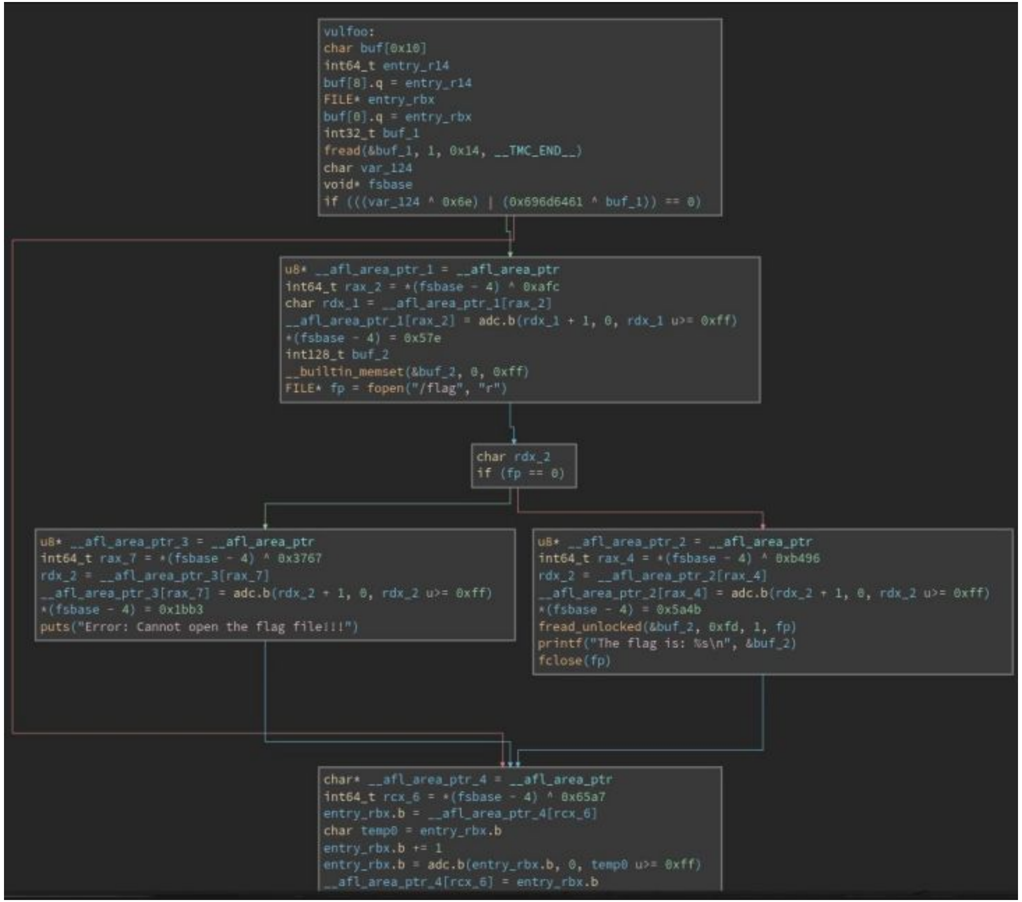
- Automated testing
  - Feed unexpected, random, or mutated inputs to a program/function that reveal a bug
- Bug discovery
  - Detect crashes, hangs, memory errors, and undefined behavior
- Have been applied to all kinds of software, e.g., web, network protocol.
  - We focus on binary/native software in this class

In its simplest form: naive fuzzing



Instrumentations





Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing

Asmita[1], Yaroslav Oliinyk[2], Michael Scott[2], Ryan Tsang[1],
Chongzhou Fang[1], Houman Homayoun[1]

[1]*University of California, Davis* [2]*NetRise*

Abstract

BusyBox, an open-source software bundling over 300 essential Linux commands into a single executable, is ubiquitous in Linux-based embedded devices. Vulnerabilities in BusyBox can have far-reaching consequences, affecting a wide array of devices. This research, driven by the extensive use of BusyBox, delved into its analysis. The study revealed the prevalence of older BusyBox versions in real-world embedded products, prompting us to conduct fuzz testing on BusyBox. Fuzzing, a pivotal software testing method, aims to induce crashes that are subsequently scrutinized to uncover vulnerabilities. Within this study, we introduce two techniques to fortify software testing. The first technique enhances fuzzing by leveraging Large Language Models (LLM) to generate target-specific initial seeds. Our study showed a substantial increase in crashes when using LLM-generated initial seeds, highlighting the potential of LLM to efficiently tackle the typically labor-intensive task of generating target-specific initial seeds. The second technique involves repurposing previously acquired crash data from similar fuzzed targets before initiating fuzzing on a new target. This approach streamlines the time-consuming fuzz testing process by providing crash data directly to the new target before commencing fuzzing. We successfully identified crashes in the latest BusyBox target without conducting traditional fuzzing, emphasizing the effectiveness of LLM and crash reuse techniques in enhancing software testing and improving vulnerability detection in embedded systems. Additionally, manual triaging was performed to identify the nature of crashes in the latest BusyBox.

a central role in IoT security, as vulnerabilities within them can jeopardize an entire system's security. Because of the role it plays in governing most aspects of a system's behavior, firmware is of particular importance. Often, firmware is comprised of numerous third-party software components that can be reused across various products, thereby amplifying concerns regarding their vulnerability; a single flaw could potentially affect multiple disparate devices that rely on shared components. Consequently, continuous analysis of these components is imperative.

1.1 Motivation

Firmware can be broadly classified into three categories: those based on modified generic operating systems (OS) like Linux, those based on real-time (RTOS) or custom operating systems, and those that do not have a formal operating system (non-OS or bare-metal). Each of these categories poses distinct challenges when it comes to security assessment [32] and often require different approaches. To that end, we focus our attention in this work on the largest subclass of OS-based firmware: Embedded Linux.

Embedded Linux-based firmware often utilizes various application-level software components, including BusyBox, Lighthttpd, Dropbear, SQLite, OpenSSL, telnet server, and various file system utilities. Consequently, there are multiple potential attack surfaces, warranting continued exploration and research. As a critical set of commonly used utility programs in embedded Linux, BusyBox [47] is a component of particular interest. It provides over 300 common Unix utilities

USENIX'2024

LLM-Fuzzer: Scaling Assessment of Large Language Model Jailbreaks

Jiahao Yu *
Northwestern University

Xingwei Lin
Ant Group

Zheng Yu
Northwestern University

Xinyu Xing
Northwestern University

Abstract

Warning: This paper contains unfiltered content generated by LLMs that may be offensive to readers.

The jailbreak threat poses a significant concern for Large Language Models (LLMs), primarily due to their potential to generate content at scale. If not properly controlled, LLMs can be exploited to produce undesirable outcomes, including the dissemination of misinformation, offensive content, and other forms of harmful or unethical behavior. To tackle this pressing issue, researchers and developers often rely on red-team efforts to manually create adversarial inputs and prompts designed to push LLMs into generating harmful, biased, or inappropriate content. However, this approach encounters serious scalability challenges.

To address these scalability issues, we introduce an automated solution for large-scale LLM jailbreak susceptibility assessment called LLM-FUZZER. Inspired by fuzz testing, LLM-FUZZER uses human-crafted jailbreak prompts as starting points. By employing carefully customized seed selection strategies and mutation mechanisms, LLM-FUZZER generates additional jailbreak prompts tailored to specific LLMs. Our experiments show that LLM-FUZZER-generated jailbreak prompts demonstrate significantly increased effectiveness and transferability. This highlights that many open-source and commercial LLMs suffer from severe jailbreak issues, even after safety fine-tuning.

process and generate vast amounts of text data with human-like fluency, making them invaluable tools for improving efficiency in various industries.

However, although LLMs have showcased their potential across various domains, the "jailbreak" issue underscores one of the pressing challenges confronting LLMs. These advanced AI systems, while capable of remarkable feats in natural language processing, also carry the risk of being exploited for harmful or malicious purposes. The issue at the core of the jailbreak problem is the potential for LLMs to generate illegal or immoral information, hate speech, and other objectionable content. Their ability to mimic human language and creativity can lead to the rapid dissemination of such content across online platforms. Addressing this problem is crucial to ensure that LLMs remain valuable tools while mitigating their potential for misuse and harm in the digital landscape.

At the moment, the manual effort of red teaming represents a promising approach to address the jailbreak issue associated with LLMs (e.g., [20]). Red teams, comprising experts explicitly tasked with appraising the security and ethical robustness of LLMs, play a pivotal role in uncovering vulnerabilities and identifying possible misuse scenarios. However, it is crucial to note that this strategy suffers from scalability issues. The constantly evolving landscape of LLM technology, coupled with its extensive adoption, places an ongoing demand on red team researchers to continually create a more diverse set of LLM-specific test cases. This underscores the challenge of relying solely on manual efforts to keep pace with the rapid

USENIX'2024

Other Advanced Software Security methods

- Address Sanitization
- Static Analysis
- Dynamic Analysis
- Symbolic Execution
- Concolic Execution
- Taint Analysis
- ...

Final CTF Schedule

- 05/07/2026, Thursday
  - 3.00 PM - 6 PM
- Location
  - CCSB 1.0410
- Homework 10
  - Due on 05/07/2026 before the exam
- Course Evaluation Deadline
  - 05/10/2026
  - We are at **33%**
  - **20 mark bonus for everyone at 90%**