CS 489 / 698: Software and Systems Security

Module: Common Vulnerabilities

Lecture: weird machine

Meng Xu *(University of Waterloo)*

Fall 2024

Outline

1. Introduction
2. A tale of two state machines
3. Defining security

Based on paper

Weird Machines, Exploitability, and Provable Unexploitability

By *Thomas Dullien* published in 2017 when he was in Google Project Zero.

Why this paper?

Why this paper?

It attempts to formalize a concept that has been intuitively known for quite a while in the community of security practitioners, i.e., both by the hackers and the researchers...

Why this paper?

It attempts to formalize a concept that has been intuitively known for quite a while in the community of security practitioners, i.e., both by the hackers and the researchers…

… and that concept is called “exploit”.

What is an exploit?

What is an exploit?

- Magic
- Access (mostly unauthorized)
- Controls the instruction pointer (e.g., EIP/RIP register)
- A program does something it is not supposed to do
- I can recognize it when I see it

They are not technically wrong, but are clearly ill-defined for academic research purposes.

Why do we bother to define it?

Why do we bother to define it?

We need to make justifications in the real-world that depends on the concept of "exploits":

- Mitigation strategies
  - e.g., difficulty of exploitation vs performance
  - e.g., difficulty of exploitation vs programmability
  - e.g., difficulty of exploitation vs complexity
- Exploitability of software/hardware defects
  - e.g., does the Rowhammer bug create a big security problem?
  - e.g., can the Spectre bug be used to launch general attacks? If yes, how?

The MitiGator

Raising the bar on exploitation until no more exploits can be seen

Copyright: The MitiGator animation

The MitiGator

Raising the bar on exploitation until no more exploits can be seen

Copyright: The MitiGator animation

The MitiGator

Raising the bar on exploitation until no more exploits can be seen

Copyright: The MitiGator animation

Learn principles, not examples

Learn principles, not examples

An important message conveyed by this paper (which is also a message I want to share with you), is that exploitation IS NOT a "bag of tricks".

Learn principles, not examples

An important message conveyed by this paper (which is also a message I want to share with you), is that exploitation IS NOT a “bag of tricks”.

In security courses (including this one), we teach

- Stack smashing, buffer overflows, heap exploitations
- SQL injection, XSS, etc
- ASLR, CFI, sandboxing, etc.

It is important to remember that there is a more fundamental principle behind these examples — *exploitation is all about entering and programming a weird machine*.

Outline

1. Introduction
2. A tale of two state machines
3. Defining security

Behind an “exploit”

By just saying “I exploited something”, you are conveying at least two messages:

- There exists some software running on top of some hardware
- There are “defects” in either the software or hardware (or both).

What is software?

What is software?

A software is an emulator for a finite-state machine (FSM) we would like to have but we don't.

What is software?

A software is an emulator for a finite-state machine (FSM) we would like to have but we don’t.

Instead, we only have a general-purpose CPU which is designed to model a huge spectrum of FSMs.

What is software?

A software is an emulator for a finite-state machine (FSM) we would like to have but we don’t.

Instead, we only have a general-purpose CPU which is designed to model a huge spectrum of FSMs.

Hence, the reason we develop software is to confine the CPU to follow and only follow the FSM we intend to have.

The intended finite-state machine (IFSM)

The state machine we want to have is called the “intended finite-state machine” (IFSM).

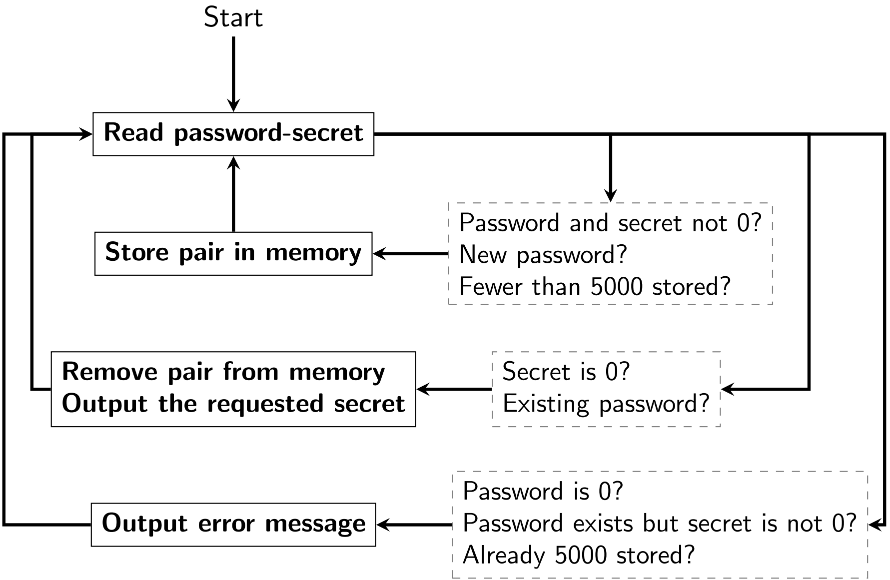
- It is usually not explicitly specified
- It is “perfect” by design — fully implements our intentions
- It cannot, by definition, have security problems.

A concrete example: a secret-keeping machine

The machine has the following functionalities:

- Reads a password / secret (p, s) from a user and remembers it.
  - NOTE: neither p nor s can be 0 (0 is reserved as an error code)
- Given a password (p) that exists in the memory, the machine returns a previously-stored secret (s) and forget both.
- The machine will not need to store more than 5000 such pairs.

IFSM diagram



Start

Read password-secret

Store pair in memory

Password and secret not 0?
New password?
Fewer than 5000 stored?

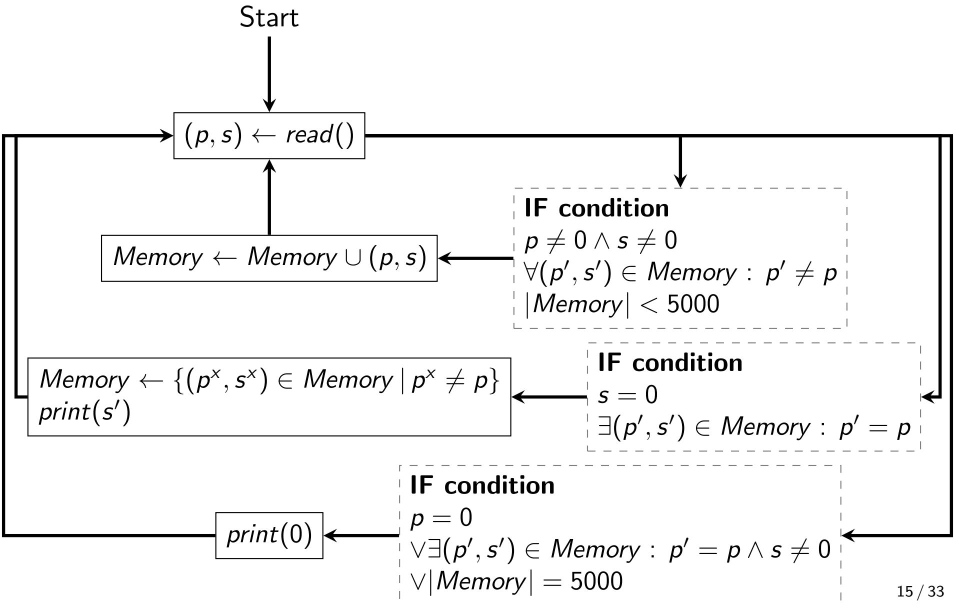
Remove pair from memory
Output the requested secret

Secret is 0?
Existing password?

Output error message

Password is 0?
Password exists but secret is not 0?
Already 5000 stored?

IFSM diagram



Start

$(p, s) \leftarrow read()$

$Memory \leftarrow Memory \cup (p, s)$

IF condition

$p \neq 0 \wedge s \neq 0$

$\forall (p', s') \in Memory : p' \neq p$

$|Memory| < 5000$

$Memory \leftarrow \{(p^x, s^x) \in Memory \mid p^x \neq p\}$

$print(s')$

IF condition

$s = 0$

$\exists (p', s') \in Memory : p' = p$

$print(0)$

IF condition

$p = 0$

$\vee \exists (p', s') \in Memory : p' = p \wedge s \neq 0$

$\vee |Memory| = 5000$

IFSM formalization

The set of all *Memory*, denoted as $\mathcal{M}$, can be formally defined as

$$\mathcal{M} = \left\{ \begin{array}{l} \emptyset \\ \{(p_1, s_1)\} \\ \ldots \\ \{(p_1, s_1), \ldots, (p_{5000}, s_{5000})\} \end{array} \middle| \begin{array}{l} p_i, s_i \in \{0,1\}^{32} - \{0\} \\ p_i \neq p_j \end{array} \right\}$$

FSM quick recap

An FSM can be defined by a 7-tuple: $(Q, i, F, \Sigma, \Delta, \delta, \sigma)$

- Q: Set of states
- i: The initial state
- F: The set of final states
- Σ: The input alphabet
- Δ: The output alphabet
- δ: State transition function $\delta : Q \times \Sigma \rightarrow Q$
- σ: Output mapping function $\sigma : Q \times \Sigma \rightarrow \Delta$

IFSM formalization — what we intend to have

The IFSM of our secret-keeping program can be defined as:

- Q: $\{A_M, M \in \mathcal{M}\}$
- i: $A_\emptyset$
- F: $\emptyset$
- Σ: $\{(p, s) \mid p, s \in \{0, 1\}^{32}\}$
- Δ: $\{0, 1\}^{32}$
- δ: $A_M \times (p, s) \rightarrow A_M \mid A_{M \cup (p,s)} \mid A_{M-(p,s)}$
- σ: $A_M \times (p, s) \rightarrow s' \mid 0$

What we actually have: a realistic CPU

The Cook-and-Reckhow RAM machine

- 2^{16} memory cells each holding a 32-bit value
- 7 CPU registers (r_0 to r_6)
- A small set of instructions
  - Constant: `LOAD(`C`,` r_d`)`
  - Register operations: `ADD(`r_{s1}`,` r_{s2}`,` r_d`)`
  - Register operations: `SUB(`r_{s1}`,` r_{s2}`,` r_d`)`
  - Memory read: `ICOPY(`r_p`,` r_d`)`
  - Memory write: `DCOPY(`r_d`,` r_s`)`
  - Control flow: `JNZ/JZ(`r`,` l_z`)`
  - Environment IO: `READ(`r_d`)`
  - Environment IO: `PRINT(`r_s`)`
- Harvard architecture (program is provided and external to RAM)

CPU FSM formalization — what we actually have

The FSM of a general-purpose CPU can be defined as:

- Q: $(q_1, ..., q_{2^{16}}) \times (r_0, ..., r_6) \times p_i$ where $q_i, r_i \in \{0,1\}^{32}$, $p_i \in P$
- i: $q_i = 0, r_i = 0, p_i = P_0$
- F: $\emptyset$
- Σ: CPU Instruction Set $\{I\}$
- Δ: $\{0,1\}^{32}$
- δ: $Q \times I \rightarrow Q'$
- σ: $Q \times I \rightarrow (e \in \Delta)$

From spec to execution: a series of refinement

We want to translate our IFSM S_{spec} into our CPU FSM $S_{execution}$.

From spec to execution: a series of refinement

We want to translate our IFSM S_{spec} into our CPU FSM $S_{execution}$.

It is actually a multi-stage process, involving (non-exhaustively)

$$S_{spec} \sqsupseteq S_{language} \sqsupseteq S_{machine} \sqsupseteq S_{execution}$$

From spec to execution: a series of refinement

We want to translate our IFSM S_{spec} into our CPU FSM $S_{execution}$.

It is actually a multi-stage process, involving (non-exhaustively)

$$S_{spec} \sqsupseteq S_{language} \sqsupseteq S_{machine} \sqsupseteq S_{execution}$$

- $S_{spec} \not\sqsupseteq S_{language}$: software bug, blame the developer
- $S_{language} \not\sqsupseteq S_{machine}$: compiler bug, blame the compiler
- $S_{machine} \not\sqsupseteq S_{execution}$: hardware bug, blame the machine

Outline

1. Introduction
2. A tale of two state machines
3. Defining security

Bug $\implies$ exploits?

Does having a bug in the refinement chain always imply a security issue (a.k.a., an exploit)?

What is security?

What is security?

Security is properties of the IFSM that we want to hold in the presence of an adversary with a specific attack model.

Security of our secret-keeper

Security of our secret-keeper

Informally, we want to ensure that anyone who interact with our program needs to know (or guess) the right password in order to obtain the stored secret.

Put in a different way, the best way to attack our program to extract some secret is to guess the password.

Security of our secret-keeper

Formally, we want the security property to hold at our IFSM:

$$Pr[s \in O_{IFSM}] \leq \frac{|I_{attempt}|}{2^{32}}$$

Security of our secret-keeper

Formally, we want the security property to hold at our IFSM:

$$Pr[s \in O_{IFSM}] \leq \frac{|I_{attempt}|}{2^{32}}$$

As well as at the final execution stage, after the refinement chain

$$Pr[s \in O_{execution}] \leq \frac{|I_{attempt}|}{2^{32}}$$

Security of our secret-keeper

Formally, we want the security property to hold at our IFSM:

$$Pr[s \in O_{IFSM}] \leq \frac{|I_{attempt}|}{2^{32}}$$

As well as at the final execution stage, after the refinement chain

$$Pr[s \in O_{execution}] \leq \frac{|I_{attempt}|}{2^{32}}$$

Even in the presence of an attacker with the assumed power of performing single chosen bit-flip.

The security property depends on the implementation

- Naive implementation: Simulate the *Memory* set as a flat linear array with sequential scanning

The security property depends on the implementation

- Naive implementation: Simulate the *Memory* set as a flat linear array with sequential scanning
- Clever implementation: Simulate the *Memory* set with two singly-linked lists.

The security property depends on the implementation

- Naive implementation: Simulate the *Memory* set as a flat linear array with sequential scanning
- Clever implementation: Simulate the *Memory* set with two singly-linked lists.

Conclusion: the clever implementation is actually vulnerable.

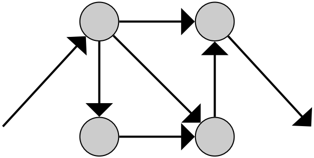
An attack on the clever implementation

1. Attacker sends (p_0, s_0), (p_1, s_1), (p_2, s_2)
2. Victim sends (p_d, s_d)
3. Attacker sends $(p_2, 0)$, $(p_1, 0)$, (p_3, s_3), (p_4, s_4)
4. Attacker gets to corrupt a single bit: flip the least significant bit for memory cell content at `b'0101` (i.e., cell `0x5`)
5. Attacker sends $(s_4, 0)$
6. Attacker sends $(12, 0)$ and obtains s_d

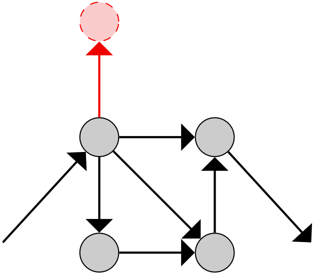
The naive implementation is secure

Please refer to the paper for the details of the proof.

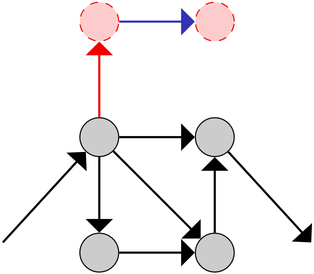
Programming the weird machine



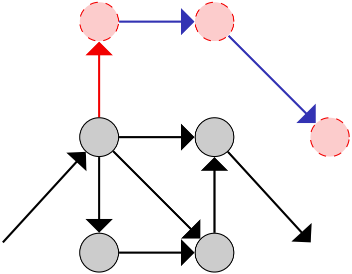
Programming the weird machine



Programming the weird machine



Programming the weird machine

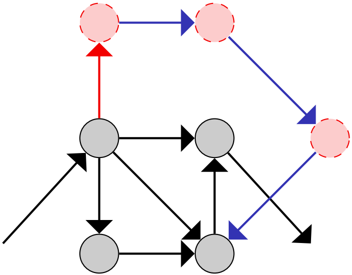


An emergent instruction set

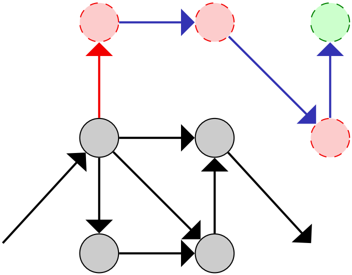
This weird machine creates an emergent instruction set that is constrained by:

- The IFSM
- The program that is refined from the IFSM
- The CPU FSM

Outcomes of weird machine programming



Reverted back to the IFSM



Reached the target state

⟨ **End** ⟩