Finding Vulnerabilities in Embedded Software

Christopher Kruegel
UC Santa Barbara
What are we talking about?

1. firmware and security
2. binary vulnerability analysis
3. vulnerability models
4. automation
Blend between real and virtual worlds

• Embedded software is everywhere
  – captured through many buzzwords
    • pervasive, ubiquitous computing
    • Internet of Things (IoT)
  – sensors and actuators
The “Internet of Things”
Increase in Lines of Code
Security Challenges

• Quantity has a quality all its own

• Vulnerability analysis
  – binary blobs (binary only, no OS or library abstractions)
  – software deeply connected with hardware

• Patch management
  – devices must be cheap
  – vendors might be long gone
Security Challenges

• Remote accessibility
  – device authentication
  – access control (pacemaker during emergency)
  – stepping stone into inside of perimeter

• Additional vulnerability surface
  – attacks launched from physical world
  – supply chain attacks

• Getting access to the firmware
BINARY VULNERABILITY ANALYSIS
Binary Analysis
Binary Analysis

- Binary code is the worst-case, common denominator scenario
Symbolic Execution

"How do I trigger path X or condition Y?"

• Dynamic analysis
  – Based on concrete inputs to application

• (Concrete) static analysis
  – "You can't" / "You might be able to"
  – based on various static techniques

• We need something slightly different
Symbolic Execution

"How do I trigger path X or condition Y?"

- Interpret the application, keeping input values abstract (symbolic)
- Track "constraints" on variables
- When a condition is triggered, "concretize" to obtain a possible input
Symbolic Execution - Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()

<table>
<thead>
<tr>
<th>State AA</th>
<th>State AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variables</td>
</tr>
<tr>
<td>x = ???</td>
<td>x = ???</td>
</tr>
<tr>
<td>Constraints</td>
<td>Constraints</td>
</tr>
<tr>
<td>x &lt; 10</td>
<td>x &gt;= 10</td>
</tr>
</tbody>
</table>
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
Symbolic Execution - Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

**State ABA**

<table>
<thead>
<tr>
<th>Variables</th>
<th>x = ???</th>
</tr>
</thead>
</table>

**Constraints**

| x >= 10  |
| x < 100  |

**Concretized ABA**

<table>
<thead>
<tr>
<th>Variables</th>
<th>x = 99</th>
</tr>
</thead>
</table>
Symbolic Execution - Pros and Cons

Pros

• Precise
• No false positives
  – with correct environment model
• Produces directly-actionable inputs

Cons

• Not easily scalable
  – constraint solving is NP-complete
  – state and path explosion
Framework for the analysis of binaries, developed at UCSB
angr Components

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
  - Control-Flow Graph
  - Data-Flow Analysis
  - Value-Set Analysis
  - Forward Symbolic Execution
  - Under-constrained SE
anqr Platform

Open Source Analysis Platform

- More than 100 KLOC
- More than 10K commits
- More than 30K downloads in 2017
- 1,600+ stars on Github
- Users in industry, academia, government
an gr - Challenges and Goals

Scalability

New Models of Malice

Precision
angr - Challenges and Goals

Scalability

Ability to compose different analyses is very powerful

New Models of Malice

Precision

UC Santa Barbara
Symbolic Execution Improvements

- Fastpath and adaptive concretization
  - when possible, analyze parts of code non symbolically

- Peephole optimization
  - replace code snippets that blow up symbolic execution

- Lazy constraint solving
  - sometimes, waiting to add more constraints makes solving easier
Constraint Solver Optimizations

- **Solution caching**
  - don’t run solver on same constraints multiple times

- **Constraint subset management**
  - break up hard constraints into subparts and solve separately

- **Expression simplification**
  - before submitting constraints, simplify

- **Expression rewriting**

![Graph showing performance comparison between Unoptimized and Base cases]
Static Analysis Support

- **Veritesting**
  - SSE to merge over multiple paths

- **LESE - loop extended sym exec**
  - intelligent loop unrolling

- **Code summarization (Dodo)**
  - automatically (and statically) summarize effect of loops / functions

- **VSA - value set analysis**
  - resolve ranges (and conditionals) without solving constraints
American Fuzzy Lop (AFL)
American Fuzzy Lop (AFL)
Combining Approaches

• angr can be used in combination with other tools

• Fuzzing excels at producing general inputs
• Symbolic execution is able to satisfy complex path predicates for specific inputs

• Key Insight
  – combine both techniques to leverage their strengths and mitigate their weaknesses
Driller = AFL + angr

Fuzzing

good at finding solutions for general inputs

Symbolic Execution

good at finding solutions for specific inputs
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
else:
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
        print "Try again..."
    return
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
        print "Try again..."
return

Test Cases

"asdf:AAAA"
"asDA:sAAA"
"aDAAA:sAAA"
"asDAL:sAAAt"
"axOO:sABBX"
"asOO:sABX"
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
print "Try again..."
return
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
        print "Try again..."
return

Test Cases

- "asdf:secret"
- "asdf:ljafe"
- "aDAA:secret"
- "aaDAA:etsf"
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
        print "Try again..."
    return

Constraints
username = ???
password = ???

password != "secret"
password == "secret"
command == "C"
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
        print "Try again..."
return
Impact of Driller

Applicability varies by program. Where it was needed, Driller increased block coverage by an average of 71%.
Impact of Driller
Failed Attempts (aka Future Research)

• State management
  – duplicate state detection

• Path selection to reach “promising” parts of the program
  – heuristics that guide analysis to areas that are more likely vulnerable
VULNERABILITY MODELS
Interesting Vulnerabilities

- Memory safety vulnerabilities
  - buffer overrun
  - out of bounds reads (heartbleed)
  - write-what-where

- Authentication bypass (backdoors)

- Actuator control
Authentication Bypass

Show me recorded video.

Please authenticate.

chris:<REDACTED>

Authentication Successful!

Here is the video.
Authentication Bypass

Show me recorded video.

Please authenticate.

service:service

Authentication Successful

Here is the video.
Authentication Bypass

Prompt

Authentication

Success

Failure
Authentication Bypass

- Prompt
- Authentication
  - Success
  - Failure

Backdoor e.g., strcmp()
Authentication Bypass

Prompt

Authentication

Backdoor e.g., strcmp()

Success

Failure

Hard to find.
Authentication Bypass

Prompt

Success

Missing!
Modeling Authentication Bypass

UC Santa Barbara

Prompt

Backdoor e.g., strcmp()

Hard to find.

Easier to find!

Authentication

Success

Failure
Input Determinism

Can we determine the input needed to reach the success function, just by analyzing the code?

The answer is NO
Input Determinism

Can we determine the input needed to reach the success function, just by analyzing the code?

The answer is YES
Modeling Authentication Bypass

Prompt

Authentication

Success

Failure

Backdoor e.g., strcmp()

Easier to find!

But how?
Finding “Authenticated Point”

- Without OS/ABI information

- With ABI information
Identify Authenticated Point

- static analysis (data references, system calls)
- human analyst fallback

"Authentication Successful!"
Compute Authentication Slice

- static analysis (program slicing)

"Authentication Successful!"
Path Collection

- authenticated path
- authenticated path
Vulnerability Detection

- can the attacker determine a concrete authenticating input via program analysis?

"AAA: XXX"
"BBB: YYY"
"CCC: ZZZ"

...
Bootloader Vulnerabilities
Bootloader Vulnerabilities

UC Santa Barbara

BL1/BootROM

BL2

BL31

BL33

Android Kernel (boot)

Android Framework/Apps (system/data)

Trusted OS (tz)

Trusted Apps

Secure World

Non-Secure World

EL 3

EL 1

EL 0

Writeable Storage

UC Santa Barbara
Two Malice Models

Memory Corruption

"Is data, read from writeable storage, used unsafely in memory operations?"

(can result in bricking, device compromise, and even TrustZone compromise!)

Unsafe Unlock

"Can the device be unlocked without triggering a user data wipe?"

(can result in data compromise)
Symbolic Taint Propagation

Taint Sources
- Writeable Storage

Multi-tag Taint Propagation
- Under-constrained Symbolic Execution

Taint Sinks
- memory dereferences
- memcpy
- loop conditions

UC Santa Barbara

63
# Results

<table>
<thead>
<tr>
<th>Bootloader</th>
<th>Sources</th>
<th>Sinks</th>
<th>Alerts</th>
<th>Memory Bugs</th>
<th>Unsafe Unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualcomm (Latest)</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Qualcomm (Old)</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NVIDIA</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>HiSilicon/Huawei</td>
<td>20</td>
<td>4</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>MediaTek</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>33</strong></td>
<td><strong>9</strong></td>
<td><strong>20</strong></td>
<td><strong>7</strong></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>
AUTOMATING VULNERABILITY ANALYSIS
From Tools Supporting Humans …

High effectiveness
Low scalability
... To Fully Automated Analysis

High scalability
DARPA Grand Challenges

Self-driving Cars

Robots
DARPA Cyber Grand Challenge

Programs!
DARPA Cyber Grand Challenge (CGC)
DARPA Cyber Grand Challenge

- CTF-style competition
- Autonomous Cyber-Reasoning Systems (CRSs) attack and defend a number of services (binaries)
- No human in the loop
- A first qualification round decided the 7 finalists
- Final event was on August 4, 2016 during DefCon
  - Shellphish came in 3rd place
- Significant cash prizes
  - 750K for qualification, 2M for win (750K for 3rd place)
CGC Results
Summary

- **Internet of Things**
  - explosive growth of devices with embedded software
  - many interesting security challenges

- **Binary analysis**
  - key tool to hunt for IOT vulnerabilities
  - delivers powerful results, but faces scalability issues
  - promising approach is to combine analysis techniques (e.g., fuzzing and symbolic execution)

- **angr**
  - UCSB open-source binary analysis software
Thank You!