ENGR-E 399/599: Embedded systems reverse engineering

Lecture 14: Reverse-engineering for security

Austin Roach
ahroach@iu.edu

April 21, 2022

Today's plan

- Common embedded systems security flaws
- Reverse-engineering approaches
- Examples

Why does embedded systems security matter?

- Embedded systems operate critical things
  - Transportation
  - Medical
  - Critical infrastructure
  - Industrial processes
  - Home automation
- Embedded system compromises can be points of entry into networks
  - Launch point for attacks behind firewalls, on private local networks, etc.

But, embedded systems security is notoriously *awful*

Why is embedded systems security so notoriously bad?

- Making embedded systems *work* is hard
  - Who wants to layer on security when it might break the system?
  - No one is going to attack my system, right?
- Standard security mitigations go unused
- Security updates not produced, or not applied
  - No update processes
  - No manufacturer support for long-lived systems
  - Risk of breaking the system, or downtime
- Attack surface gradually increased over time
  - Formerly standalone systems added to networks
- Use of home-grown services and protocols
  - Don't use vetted applications, libraries, or protocols
- Embedded systems developers and users often not taught about security

Common vulnerabilities

- Weak credentials
  - Development/debug credentials
  - Default credentials unchanged by system user
- Backdoors
  - Often for development/debug, or automated configuration
- Authentication flaws
  - Authentication bypass
  - Session hijacking
  - Replay attacks
- Memory safety vulnerabilities
  - Spatial memory safety: buffer overflows
  - Temporal memory safety: use-after-free, double free
- Command injection
- Encryption flaws
- Incorporation of unpatched software/libraries
- Firmware updates unverified/unsigned
- Lack of secure boot
- Physical attacks

Reverse-engineering approaches

Much easier to check for flaws with design information.

But, many of the reverse engineering approaches that we've discussed in this class can help with security analysis of a black-box system

First steps: Identify interfaces

Datasheet inspection

Physical inspection

- UART
- Other interfaces

Network inspection

- nmap

Define the attack surface

Static analysis: Filesystem inspection (where applicable)

- Identify applications and libraries, including version numbers
  - Create a *software bill-of-materials* (SBOM)
    - Basis for searching for known vulnerabilities (CVEs)
- Identify system configuration scripts/files
  - Look for services that are configured insecurely
- Extract stored credentials (or hashes)
  - Try to crack hashed passwords, identify weak credentials

Basic system reconnaissance can uncover many flaws

Static analysis: Building an SBOM for deeply embedded software

Best hints come from strings:

- Version strings
- Unique debug messages

But may also come from instructions/implementation if some information is already known:

- "I think this uses library X but I don't know what version"
- Identify differences between versions and compare to binary

More time consuming, but will allow you to use existing information about vulnerabilities

Static analysis: Look for flaws in extracted applications/libraries

- Look for dangerous functions
  - strcpy(), strcat(), sprintf()
  - printf() with format string controlled by attacker
  - memcpy(), read(), etc. with size controlled by attacker
  - system()
  - etc.
- Look for loops performing memory operations with attacker-controlled values
- Look for security-critical functions
  - Authentication
  - Cryptography
- etc.

There's a whole world of software bug hunting that applies here

But my program doesn't have labeled functions...

Find equivalents in deeply embedded software

- Automated function identification approaches mentioned before
- But likely a lot of manual analysis

Dynamic analysis: Probe interfaces

- Monitor communications during normal operation
- Probe for common vulnerabilities
  - Existing automated tools for things like web vulnerabilities
  - Can check for things like command injection manually
- Deploy fuzzers against the interfaces

Can find flaws without having to locate the source in firmware

Static/dynamic analysis: Inspect bootloader or firmware update code

- Firmware updates signed?
  - Using public-key cryptography, preferably
- What is the firmware update format?
- Can a modified firmware be supplied to the device?

Modifying firmware can be a path to greater system debug access, and may be a security flaw in itself.

Dynamic analysis: Physical attacks

- Glitching (voltage, optical, ...)
  - Bypass software checks, lock bits, etc.
- Side-channel analysis
  - Extract secret information
  - Memory readout
    - Including things like mask ROM

May provide more information for reverse engineering, or may be a security vulnerability in itself.

Examples

Code execution on a wifi router

- Downloaded firmware update, and extracted squashfs
- Found a 4-pin UART connection, with access to password-protected console
- Modified shadow file in filesystem to allow login using known password, allowing debug, observation of core dumps, etc.
- Searched filesystem for code calling recvfrom() and sendto(), indicating UDP services
- Found an administration service with weak encryption of its commands
- Exploited this service to issue commands and exploit a stack overflow

`https://blog.senr.io/blog/cve-2017-9466-why-is-my-router-blinking-morse-code`

Code execution on an IP camera

- Downloaded firmware update, and extracted JFFS2 filesystem
- Identified services using nmap
- Enabled ssh over a web interface, and logged in with default credentials to allow debugging
- Searched code for any functions that wrote to stack buffers
- Identified a looped write in one service without proper bounds checking
- Created a ROP payload to access a shell

`https://blog.senr.io/devilsivy.html`

Command injection in a wifi router

- Downloaded firmware update and extracted squashfs
- Identified relevant server binaries
- Identified user-controlled arguments passed to system()
- Incorrect authentication check allowed commands to be provided without authentication

`https://web.archive.org/web/20210614121453/http://www.devttys0.com/2015/04/hacking-the-d-link-dir-890l/`

Multiple flaws in an industrial control service gateway

- Retrieved firmware update file for the device
- Built an automated testing and analysis framework to send in Modbus TCP and observe Modbus RTU
- Used a fuzzer to generate test messages
- Identified inputs that caused the device to stop responding
- Found multiple authentication, encryption, and protocol parsing flaws

`https://i.blackhat.com/USA-20/Wednesday/us-20-Balduzzi-Industrial-Protocol-Gateways-Under-Analysis-wp.pdf`

Remote code execution on PLCs

- Accessed firmware (somehow)
- Monitored network communications
- Identified functions that implemented extensions to the standard Modbus protocol
- Identified a memory-read function that could be used to bypass authentication on other commands
- Demonstrated bypass for certain security mechanisms by uploading a new project file to the controller

https://www.armis.com/research/modipwn/