Automating Analysis and Exploitation of Embedded Device Firmware

By: Malachi Jones, PhD
About Me

Education

- Bachelors Degree: Computer Engineering (Univ. of Florida, 2007)
- Master’s Degree: Computer Engineering (Georgia Tech, 2009)
- PhD: Computer Engineering (Georgia Tech, 2013)

Cyber Security Experience

- Harris: Cyber Software Engineer (2013-2014)
- Booz Allen Dark Labs: Embedded Security Researcher (2016- Present)

https://www.linkedin.com/in/malachijonesphd
Booz Allen Dark Labs is an elite team of security researchers, penetration testers, reverse engineers, network analysts, and data scientists, dedicated to stopping cyber attacks before they occur.¹

¹ http://darklabs.bah.com
Outline

- Motivation

- Background
  - Firmware Analysis
  - Automated Exploit Generation
  - Intermediate Representation (IR) Languages

- LLVM

- Architecture Independent Analysis and Exploitation

- Conclusion
Motivation

- Embedded in Society

- Critical Infrastructure
  *(Nuclear Power Plant)*

- Life Critical Systems
  *(Pace Maker)*

- Transportation Systems
  *(Jeep)*

- Financial Infrastructure
  *(Banking & Investing)*

- Internet of Things (IoT)
  *(IoT Gadgets)*

- Commercial Products
  *(Network Switch)*
Motivation

- Workhorses Behind the Embedded Scene
  - Hexagon
  - MSP 430
  - SuperH
  - MIPS
  - ARM
  - PowerPC
Motivation

Why is embedded device security difficult? (vs. gen. purpose computing)

1. Multi Architecture Support:
   - Plethora of architectures that are utilized in embedded devices versus ubiquitous adoption of x86 & x86_64 for general purpose computing
   - This often requires security tool development for each architecture
Motivation

Why is embedded device security difficult? (vs. gen. purpose computing)

2. Custom Hardware:
   - Embedded devices utilize custom and/or esoteric hardware (e.g. sensors) to perform specialized tasks
   - **Difficult to emulate custom hardware, which is often required to achieve scale for dynamic analysis**
Motivation

Why is embedded device security difficult? (vs. gen. purpose computing)

3. **Environmental Constraints**:
   - Depending on where the device is deployed, it may be constrained by mass, power, cost, or volume that can also impact performance and memory
   - *Mainstream features on general purpose devices such as ASLR or DEP may be sacrificed to satisfy environmental and/or computational constraints*
Why is embedded device security difficult? (vs. gen. purpose computing)

4. **Security as an Afterthought:**
   - Often financially and/or technically infeasible to retrofit security capabilities to an embedded system that was not originally designed for it
   - *Once deployed to target environment, embedded devices may be in operation for 10+ years. Because of (3), Moore's Law does not apply*
Objectives of Talk

- Discussion of an approach for addressing the challenge of building analysis tools that can *support multiple embedded architectures*

- Specifically, we’ll explore an approach for *decoupling architecture specifics* from the analysis by utilizing LLVM, a widely supported intermediate representation (IR) language
Firmware Analysis

Background
Background: Firmware Analysis

- **Static Firmware Analysis:**
  - Analysis of computer software that is performed without the actual execution of the software code
  - **Data Flow analysis** is a type of static analysis that can be used to understand and evaluate how “data flows” through the code paths of the program
  - **Taint analysis** is a specific application of data flow analysis that follows user controlled data to identify code paths that process that data
Taint Analysis

Can be very instrumental in identifying user-controlled vulnerable code

General Process

Step 1: Identify source data inputs that originate from user

Step 2: Follow the code paths that process (e.g. transformations and reads) the user data inputs

Step 3: Keep track of code that reads the user data

A simple example to illustrate the concept of taint analysis can be seen on the following slide
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initiValues);

int main(int argc, char *argv[]){
    int myArray [10];

    if(argc != 2)
    {
        printf("usage:Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);

    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i] ;
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
Step 1: Identify Originating User Controlled Input

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char * argv[]){
    int myArray [10];
    if(argc != 2){
        printf("usage:Expected 2 arguments... Received:%d\n", argc);
        return 1;
    }
    char * values = argv[1];
    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);
    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i];
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
```

User controlled input (via command line)
Step 2: Follow Code that Processes Data

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char *argv[])
{
    int myArray [10];
    if(argc != 2)
    {
        printf("usage: Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];
    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);
    for( int i =0; i <length; i++)
    {
        someArray[i] = (int) initializingValues[i];
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}

'values' holds a reference to user controlled data
```
Step 2: Follow Code that Processes Data

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char *argv[]){
    int myArray[10];
    if(argc != 2){
        printf("usage: Expected 2 arguments... Received:%d
",argc);
        return 1;
    }
    char * values = argv[1];
    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);
    for(int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i];
        printf("someArray[%d] = %d
",i, someArray[i]);
    }
    return 0;
}
```

Call to method that indirectly uses user controlled data
Step 2: Follow Code that Processes Data

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char * argv[]){
    int myArray [10];
    
    if(argc != 2)
    {
        printf("usage:Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char * initializingValues){
    int length = strlen(initializingValues);
    
    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i] ;
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
```

Alias of values, which is user controlled
Step 2: Follow Code that Processes Data

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char *argv[]){
    int myArray[10];
    if(argc != 2)
    {
        printf("usage: Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char * initializingValues){
    int length = strlen(initializingValues);
    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i] ;
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
```

'`strlen` function reads value
Step 3: Identify read operations

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char *argv[]){
    int myArray [10];
    
    if(argc != 2)
    {
        printf("usage:Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);
    
    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i];
        printf("someArray[%d] = %d\n",i, someArray[i],
    }
    return 0;
}
```

Read operation performed on user controlled data
Background: Firmware Analysis

- Dynamic Firmware Analysis:
  - Execution of software in an instrumented or monitored manner to garner more concrete information on behavior
  - Typically, software is executed in an instrumented emulator (e.g. QEMU) as the emulator offers fine grained execution control
  - **Emulators** also provide the ability to parallelize the analysis without the need of additional physical devices
Complications of Dynamic Analysis in Embedded Systems

- Dynamic analysis is most effective via an emulator, but *emulation of embedded devices can be non-trivial*
- Embedded devices often use many variations of esoteric hardware that have little to no documentation, which makes emulating hardware problematic
- The emulators may have limited support for the firmware’s processor architecture or the particular version of the processor
Approaches to address emulation problem (Not exhaustive)

- **Manual Static Analysis of Native Binary**
  - Popular approach that can require a significant amount of manual human analysis
  - Much manual effort spent identifying & filtering out false-positives

- **Event Driven Dynamic Analysis Framework: (Avatar) [1]**
  - Firmware code is executed inside an emulator.
  - Any I/O access is then intercepted and forwarded to the physical device

- **Firmware Adaption [2]**
  - Extracting limited parts of firmware code to emulate it in a generic emulator
  - The focus is typically on user code that does not require significant I/O access or system calls
Background: Firmware Analysis

- Static vs. Dynamic Analysis
  - Static analysis scales well and can provide better code coverage
  - Dynamic analysis can uncover more “actual” vulnerabilities because only code paths that generate unexpected behavior during execution are analyzed
  - A potential code path marked as vulnerable during static analysis may not be reachable during actual execution
  - Static analysis requires that you know the type of vulnerability that you want to look for (e.g. buffer overflow and integer underflow)
Automated Exploit Generation

Background
Automated Exploit Generation (AEG)

- Given a program, *automatically find vulnerabilities and generate exploits for them*.
- One of the core objectives in DARPA’s *Cyber Grand Challenge*
Background: Automated Exploit Generation

- Steps for AEG [3]
  1. **Bug-finding**: Perform dynamic binary analysis to discover unsafe execution states
  2. **Exploit Generation**: For a specified unsafe execution state, generate a candidate exploit input (e.g. return-to-stack and return-to-libc)
  3. **Verification**: Feed in the exploit input into program to verify that control flow was altered in a desirable manner (e.g. spawn a shell)
Commonly used bug-finding techniques for AEG

- **Fuzzing**: Generate random permutations of a given input and monitor the program for crashes.
- **Symbolic Execution**: Analysis of a program to determine the necessary inputs needed to reach a particular code path. Variables modeled as symbols.
- **Concolic Execution**: Used in conjunction with symbolic execution to generate concrete inputs (test cases) from symbolic variables to feed into program.
- **Selective Symbolic Execution**: Fuzzing + Selective Concolic Execution

*Approach used by the CGC teams that include Shellphish [4]*
Background: Automated Exploit Generation

- **Example:** Symbolic Execution

```c
int x;
mksymbolic(x);

if (x > 0) {
    ...
} else {
    ...
}

if (x > 10) {
    ...
} else {
    ...
}
```

Example taken from the following publication: Symbolic Crosschecking of Data-Parallel Floating-Point Code (2014)
Complications with AEG (not exhaustive)

- Not all bugs are exploitable (e.g. may not be able to alter control flow in a desirable manner)
- Not all exploits are reliable (e.g. exploit requires an unlikely execution state)
- Discovering the exploitable path among an infinite number of feasible paths is non-trivial
- *Requires dynamic analysis, which is also non-trivial for embedded systems*
Intermediate Representation (IR) Languages

Background
**Formal Definition:** The language of an abstract machine designed to aid in the analysis of computer programs\(^2\)

**IR Languages** (Not Exhaustive):

1. Java Byte Code
2. Microsoft’s Common Intermediate Language (shared by .NET Framework compilers)
3. ESIL\(^3\) (radare2 disassembler)
4. BAP [5] (Binary Analysis Platform)
5. REIL [6] (Static Code Analysis)
6. SWIFT\(^4\)
7. LLVM [7] (Compiler Optimization)

---

(3 https://radare.gitbooks.io/radare2book/content/esil.html)  
(4 https://github.com/apple/swift/blob/master/docs/SIL.rst)
Background: IR Languages

- **IR Utilization in Disassemblers**
  - An approach that disassemblers (e.g. IDA Pro, Binary Ninja, and radare2) utilize is to convert the binaries to IR for control flow and data flow analysis.
  - For example, **radare2** supports the following architectures: 6502, 8051, CRIS, H8/300, LH5801, T8200, arc, **arm**, avr, bf, blackfin, xap, dalvik, dcpu16, gameboy, i386, i4004, i8080, m68k, malbolge, **mips**, msil, msp430, nios II, **powerpc**, rar, sh, snes, sparc, tms320 (c54x c55x c55+), V810, **x86-64**, zimg, risc-v.
  - Instead of creating an analysis tool for each architecture, **radare2** performs analysis on its custom IR, ESIL (Evaluable Strings Intermediate Language).
  - Example x86 to ESIL Translation:
    ```
    mov eax, [0x80480] 0x80480,[],eax,=, #8
    ```
    
    (4 https://github.com/radare/radare2)
LLVM is a common infrastructure to implement a broad variety of compiled languages that include:

- The family of languages supported by GCC (e.g. C, and C++)
- Java
- .NET
- Python (via Cpython)

- Typical use case
  1. Translate programming language (e.g. C) to llvm IR *(Front end)*
  2. Perform compiler optimizations on llvm IR *(Optimization)*
  3. Translate llvm to target machine language, e.g. x86 *(Back end)*

Example “hello world” LLVM IR

```llvm
; Declare the string constant as a global constant.
@str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {
  ; i32()*
  ; Convert [13 x i8] to i8 *
  %cast210 = getelementptr [13 x i8], [13 x i8]* @str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

(7 http://llvm.org/docs/LangRef.html)
Supported back end targets include:

- x86 & x86_64
- ARM
- MIPS
- PowerPC
- Hexagon

**Back end code is typically maintained by the processor’s designers** (e.g. Intel maintains the x86 & x86_64 llvm back end)
Analysis Libraries

- One of the core functions of LLVM is to perform optimizations (e.g. eliminate dead code and redundant stores) on its IR to produce efficient code.

- It uses a powerful set of libraries written in C++ to analyze the code to identify optimizations.

  *These libraries can also be used for static analysis to find potential vulnerabilities*.

- **Example:** We can perform loop analysis on any llvm instruction to determine the following:
  - If the instruction is in a loop
  - What are the exit conditions for the loop (e.g. $i<10$)

  Could be useful in identifying buffer overflows.
Architecture Independent Analysis and Exploitation
So how can we utilize LLVM to analyze & exploit firmware?

- Build a tool that can perform automated static analysis on the IR to find potential bugs
- In particular, we can exploit the fact that static analysis can provide us with more comprehensive code coverage
- Bugs that we may be interested in identifying include use-after-free, buffer overflow, and buffer underflow
- Static Analysis Example
  - Suppose we have a binary ‘simpleArray’ that has a potential buffer overflow vulnerability in one of its functions
  - The vulnerable code in its C representation can be seen on the next slide
# include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int * someArray, char * initValues);

int main(int argc, char *argv[]){
    int myArray [10];

    if(argc != 2)
    {
        printf("usage:Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}

int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);

    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i] ;
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
Architecture Independent Analysis and Exploitation

- Snippet of llvm ir (Static Analysis Example)

```llvm
define i32 @initializeArray(i32* %someArray, i8* %initializingValues) #0 {
  %1 = alloca i32*, align 8
  %2 = alloca i8*, align 8
  %length = alloca i32, align 4
  %i = alloca i32, align 4
  store i32* %someArray, i32** %1, align 8
  store i8* %initializingValues, i8** %2, align 8
  %3 = load i8** %2, align 8
  %4 = call i64 @strlen(i8* %3) #3
  %5 = trunc i64 %4 to i32
  store i32 %5, i32* %length, align 4
  store i32 0, i32* %i, align 4
  br label %6

  ; <label>:31                                 ; preds = %6
  ret i32 0
}
```
Static Analysis Example

- **Objective is to identify buffer overflows that occur on fixed size arrays**
- Next few slides will demonstrate how we can use our tool to accomplish this
Buffer Overflow Detection Example

```
!tbaa !4
------------------------Loop Analysis------------------------
**User performs write in loop * store i32 %9, i32* %10, align 4, !tbaa !4" **
"----------Values that the target value : * %lftr.wideiv = trunc i64 %indvars.iv
 to i32' is dependent on----------"
"----------Values that the target value : * %5 = add i32 %4, -1' is dependent on-
----------"
Value Dependency: * %1 = tail call i64 @strlen(i8* %initializingValues) #4"

(3.1) Performing loop analysis
"----------Values that the target value : * %lftr.wideiv = trunc i64 %indvars.iv
to i32' is dependent on----------"
"----------Values that the target value : * %5 = add i32 %4, -1' is dependent on-
----------"
Value Dependency: * %1 = tail call i64 @strlen(i8* %initializingValues) #4"

(3.2)Loop exit condition: * %exitcond = icmp eq i32 %lftr.wideiv, %5"

(3.3) User controls the following operand of exit condition: * %5 = add i32 %4,
-1"

Therefore, exit condition is controlled by user
```
Buffer Overflow Detection Example

```c
int initializeArray(int * someArray, char *initializingValues){
    int length = strlen(initializingValues);
    for( int i =0; i <length; i++) {
        someArray[i] = (int) initializingValues[i];
        printf("someArray[%d] = %d\n",i, someArray[i]);
    }
    return 0;
}
```

Value Dependency:  "  %1 = tail call i64 @strlen(i8* %in,...

(3.2) Loop exit condition:  "  %exitcond = icmp eq i32 %lftx.wideiv, %5"

(3.3) User controls the following operand of exit condition:  "  %5 = add i32 %4 , -1"

Therefore, exit condition is controlled by user

User controls operand ‘length’ of exit condition

User Controlled operand of exit condition
Buffer Overflow Detection Example

Out of bounds write detected (myArray)
# SimpleArray.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int initializeArray(int *someArray, char *initializingValues)
{
    int length = strlen(initializingValues);

    for( int i = 0; i < length; i++) {
        someArray[i] = (int) initializingValues[i];
    }

    return 0;
}

int main(int argc, char *argv[])
{
    int myArray [10];

    if(argc != 2)
    {
        printf("usage:Expected 2 arguments... Received:%d\n",argc);
        return 1;
    }
    char * values = argv[1];

    initializeArray(myArray, values);
    return 0;
}
```

Buffer overflow occurs if user passes in a string with length > 10

Overflowed buffer
Architecture Independent Analysis and Exploitation

- Open Source Analysis Tool (Klee)
  - **Klee** is a popular analysis tool that takes as input llvm bitcode
  - Applied to all 90 programs in the GNU COREUTILS utility suite, which forms the core user-level environment installed on most Unix systems [9]
  - When program execution branches based on a symbolic value, klee follows both branches at once, maintaining on each path a set of constraints called the path condition
  - *When a path terminates or hits a bug, a test case can be generated by using the current path condition to find concrete values that can generate the bug*
Conclusion

- Automated vulnerability analysis tools have the potential to allow the larger embedded community to conduct effective analysis, at scale, that has historically been limited to a small group of security experts.

- However, there are some challenges (e.g. hardware emulation and multi-architecture support) that will need to addressed before the potential can be realized.

- In this talk, we’ve discussed an approach to address the multi-architecture support challenge by utilizing LLVM IR.
References


Questions?

- Contact Information
  - Email: jones_malachi@bah.com
  - LinkedIn: https://www.linkedin.com/in/malachijonesphd