IoT Firmware Vulnerability Hunting with the angr Symbolic Execution Engine

Benjamin Gilman
Washington University in St. Louis
bengilman@wustl.edu

Abstract

Internet of Things (IoT) devices are increasing in ubiquity every year, and their vulnerabilities are becoming more critical. This issue has gained much attention in the past decade, leading to the research and development of tools to help combat this issue. While the toolset for firmware analysis is now much more extensive than it was, it can always be expanded and improved further.

The goal of this project is to gain a better understanding of the existing toolset, examine how it might be improved, and to use it to search for vulnerabilities in IoT firmware.

This work led to two outcomes. The first is the identification and improvement of several issues in SaTC, a popular firmware taint checking tool. Most significantly, I refined the modularity of the tool. This improves its usability and allows the tool to be applied to a wider variety of firmware, rather than just those with web-based front-end interfaces. I also extended the number of sink functions in its taint checking implementation, which resulted in finding several new paths. Lastly, there were some code reuse issues in the SaTC implementation which could be refactored to improve maintainability and efficiency.

The second is the creation of a plugin extension for the Ghidra reverse engineering framework, called GhidraVSA. This plugin allows the user to quickly perform value set analysis of a binary from within the Ghidra GUI.

Through the development of these two tools, I explored the limitations of the angr symbolic execution framework. I found that while it is a useful and powerful tool, it fails to scale. While the SaTC taint check engine terminates in simple cases, for real world software it does not complete in reasonable time.

1 Introduction

Internet of Things (IoT) devices are more commonplace than ever due to the wide availability of internet connection. While they provide extraordinary convenience, and have revolutionized the way we live, they also present a serious security risk. The fast innovation in the IoT space has left many security flaws in its wake, and due to the prevalence of these devices, these vulnerabilities pose a significant threat.

This issue has gained the attention of researchers, leading to the development of many tools to help in firmware analysis. These tools attempt to automate the analysis of firmware, exposing vulnerabilities that exist in modern IoT devices, and aiding in their correction. They have proven to be effective in identifying vulnerabilities and improving the security of these devices. However, further research and development are needed to enhance the efficiency and effectiveness of these tools.

The goal of this project is to gain a deeper understanding of the current state of firmware analysis technology and its limitations, add to and improve this technology, and use it to search for vulnerabilities in IoT firmware.

While there are many tools available, this project focuses specifically on two. The first is SaTC which, while proven to be effective, has several issues that limit its usability. In this work I focus on these issues. The second tool is GhidraVSA, a utility of my own design. It aids in static analysis by providing a simple interface for performing value set analysis from within the GUI of Ghidra – a popular open-source reverse engineering framework. Both SaTC and my own tool leverage a symbolic execution framework called angr as their binary analysis engine.

2 Background

Since there are many firmware analysis utilities available, this section will provide an overview of a relevant subset.

One such tool is Binwalk, a tool for analyzing and extracting firmware images. Since firmware images are monolithic files, a tool is needed to extract the filesystems contained within for further analysis.

Another tool is SaTC (Shared-keyword aware Taint Checking) [1], a static analysis tool which uses taint analysis techniques to track the flow of data within the firmware and identify potential vulnerabilities. This tool will be discussed in further detail in Section 4.

There are also two relevant frameworks which, while not restricted to firmware analysis, are relevant to both SaTC and GhidraVSA.

The first is Ghidra – an open-source reverse engineering framework originally developed by the NSA. The main feature of Ghidra is its decompiler which can translate machine code for many different architectures into a C-like representation called p-code. Ghidra has a rich user interface with many tools for binary analysis, and it is also extensible, allowing its functionality to be augmented either via a comprehensive scripting API or installable plugins.

The second is angr, an open-source Python library for analyzing binaries of many different architectures. Its main function is symbolic execution. This is a program analysis technique which executes software with symbolic inputs rather than concrete inputs, which can take on any value that satisfies a set of constraints. Variables dependent on the symbolic inputs are symbolic as well. Execution paths of the program are explored, and each branch adds additional constraints to the variables.

3 GhidraVSA

3.1 Motivation

Value Set Analysis (VSA) is a technique that determines the possible values that a variable can assume throughout the execution of a program. This is often accomplished through symbolic execution techniques. VSA is helpful for general program analysis, and specifically the detection of vulnerabilities. It can determine if a certain sensitive variable is underconstrained such that it causes a vulnerability, or to ascertain whether the conditions needed to reach a vulnerable section of code are feasible.

The plugin for Ghidra that I developed, named GhidraVSA, provides a graphical interface through which a user may employ VSA to aid in the analysis of a given binary. While Ghidra's decompilation is helpful for reverse engineering binaries, users must still manually determine the values that a given variable can assume at a given point in execution, which can be time consuming and difficult. By implementing VSA functionality into Ghidra, it streamlines the analysis workflow, saves time, and increases efficiency.

3.2 Implementation

GhidraVSA implements its user interface in Java, through which it collects its inputs that are then passed to a Python script that leverages the angr symbolic execution framework. The user selects an instruction – the point in execution at which the possible values for a given variable are determined – by either clicking on the instruction in Ghidra's UI or manually entering the address of the instruction. The variable itself can either be the value of a register or a memory address.

The arguments are passed to the Python script via a file. The script uses the angr API to generate a control flow graph (CFG), a graph in which the nodes are "basic blocks" of instructions that are connected according to the control flow of a binary. For instance, a basic block may consist of a few instructions that end with a conditional jump instruction. This block might be connected to two other blocks, one beginning with the subsequent instruction, while the other beginning with the instruction to which execution jumps if a condition passes.

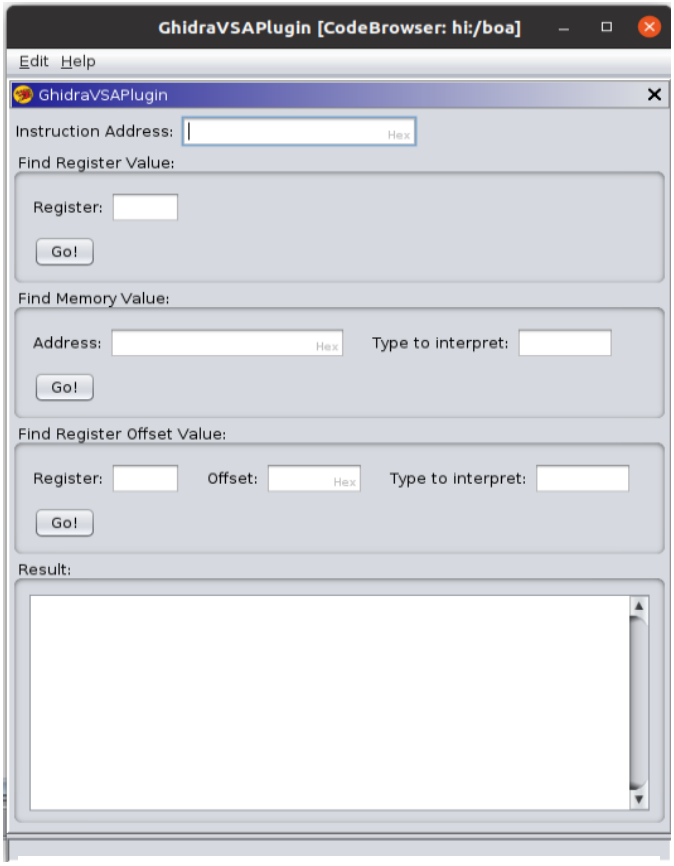

Figure 1: The GhidraVSA tool

This CFG can be used as the basis for a value flow graph (VFG). Once the control flow graph of the program is constructed, angr identifies merge points in the CFG and cuts loopback edges, resulting in an acyclic CFG. Through symbolic execution, these merge points are converted into fix points, which are final invariant states of each basic block. There may be multiple nodes in the VFG for a certain basic block due to the different control flows that may lead to that block, and there may be several final states for each of these VFG nodes. States are stored in a SimState object that represents the state of a program during a particular point in symbolic execution. These SimState objects store symbolic constrained variables for each register and memory location. Through the VFG, we can determine the possible values a variable may hold at any given point in execution.

The granularity of this analysis is limited by the size of the basic blocks. For example, if a block contains instruc-

tions with addresses ranging from 1-10, it is not possible to determine the value of a variable after executing instruction 5, only the final state of that variable after executing instruction 10. However, since there are no control flow branches within a basic block, each contains deterministic code. Therefore, the value of a variable at a point in the middle of the block is usually what it will be at the end of a block. If not, its value can easily be determined by its value in the previous block and the few instructions between that may alter it.

The GhidraVSA Python script gets the SimState associated with the block that contains the instruction the user enters and reads the symbolic variable representing the register or memory location requested. The constraints of the symbolic variable determine what values that variable may assume, and these are passed back to the user interface and displayed to the user.

4 Improvements in SATC

Works prior to SaTC have implemented taint checking as a method of detecting vulnerabilities. In this technique data inputs are marked as "tainted" since they are given by a potentially malicious user. The flow of this data is tracked throughout the program, with variables that depend on tainted data marked as tainted themselves. If the tainted data reaches a sensitive function without being sanitized, there is a potential vulnerability.

SaTC [1] proposed a novel approach to detecting firmware vulnerabilities, their key insight being that strings are commonly shared between front-end web interfaces and back-end binaries. The front-end files are scanned for keywords, and the existence of these keywords in the back-end files are used as the entry point for taint analysis. They also improve upon the efficiency of previous taint checking engines by first performing a faster, imprecise taint check, which guides a second more precise analysis.

SaTC is a powerful tool, yet there is always room for improvement. Particularly, I identified three areas in which SaTC could be enhanced: its modularity, completeness in its identified sink functions, and code reuse.

3.1 Modularity

3.1.1 Motivation

SaTC's pipeline has three steps: Front-end analysis, coarse taint analysis using Ghidra, and fine taint analysis using angr. This pipeline is effective, but it is somewhat monolithic. While command line arguments to SaTC allow the user to execute all parts up to a certain point, they do not allow the user to run just the latter sections.

SaTC's taint analysis engine improves on previous engines, and it would be valuable to apply it to more than just the firmware with web-based interfaces. Isolating it from the rest of the pipeline would allow it to be run on any binary. Additionally, each part of the SaTC pipeline is a lengthy and error-prone process. The ability to rerun individual sections is helpful for debugging and usability.

3.1.2 Implementation

The key information that front-end analysis stage of the pipeline returns is a list of keywords and the "border binaries" in which they are found. The second stage of the pipeline runs a Ghidra script several times, once for each border binary. The script finds all instances of each keyword and determines whether there is a potential taint from these locations.

To effectively decouple the Ghidra step from the keyword step, I added the capability to provide user input that points the Ghidra script to a file containing the list of keywords generated as well as a list of paths to the binaries to analyze. Since these parameters were outputs of the first step of the pipeline, when that step is not run, these parameters will not exist in memory so they must be read from elsewhere.

To expand the use of this taint check engine beyond firmware with web interfaces to all binaries, it is necessary to remove the dependance on keywords entirely. To accommodate this, I added functionality such that instruction addresses can be passed to the script, which can be used as the starting point for taint analysis instead of the addresses of references to the keyword strings.

The Ghidra analysis stage starts with the function containing a reference to a keyword or a target instruction and finds all functions that are called from within that function. It continues in each of these functions recursively until there are no more calls, or it reaches a potentially vulnerable sink function such as `system()` or `strcpy()`. If a path is found, the addresses of each call and callee along the path is written to a file, and this file is used as input to the next step of the pipeline.

The third and final step of the pipeline performs a fine-grained taint analysis using angr. It uses the output of the coarse taint analysis in the previous step to guide its exploration of the target program. Therefore, it is not possible to fully decouple the second and third steps of the pipeline – the output of the former is necessary for the latter. However, it is not necessary to run both at the same time. The output of the second step is stored in a directory, and I added the capability to run just the third step of the pipeline by providing the location of this directory as user input. Since the third step of the pipeline is the lengthiest computation and the most prone to crashes, the ability to isolate it is advantageous. There is also a benefit in that it can be run on only the border binaries of interest, rather than all that were detected by the earlier stages of the pipeline, allowing analysis to be focused as the user pleases.

3.2 Additional Sink Functions

3.2.1 Motivation

SaTC is designed to detect taint from an input to any of several sink functions, which are functions that present a security vulnerability if unconstrained user input is allowed to reach them. However, there are several such functions that are missing from their implementation. Adding additional sinks will allow SaTC to detect a wider range of vulnerabilities.

3.2.2 Implementation

The set of sink functions is defined in several points in SaTC's implementation. Each Ghidra script has its own, which consists of a simple list of the names of functions. In the taint analysis engine, there is also a list of the sinks, as well as a function for each that defines how taint is detected for that sink. Since there is only a vulnerability present if the taint reaches a particular argument to the sink, these functions are necessary for each sink to determine whether the argument to which the taint is sensitive. This is accomplished by determining which register contains the tainted data, and whether that register corresponds to the sensitive argument.

To expand the set of sink functions that may be detected with SaTC, all these lists must be updated to contain the newly added sink, and a representative function for that sink must be added as well. This is mostly trivial, and the implementation of SaTC can easily be extended to handle a much larger set of sinks.

3.3 Code Reuse

3.3.1 Motivation

The Ghidra step of the pipeline consists of several Ghidra scripts, of which the user may choose to run all or a subset. They largely perform the same task, with slight differences for various situations. For instance, there is one to detect buffer overflow vulnerabilities and another to detect command injection vulnerabilities. These scripts are almost entirely the same, the only differences being their list of sink functions. There are other scripts with slightly larger differences, but the fact remains that most of the code is repeated between them. This makes maintaining SaTC more difficult, as changes to one of these files must be made to all. This is difficult to track and prone to mistakes. Additionally, if it is desired to run more than one of these scripts, the main loop that exists in each must run multiple times, which is suboptimal.

3.3.2 Implementation

Since each of these scripts performs a slightly different function, it is important to maintain that modularity when combining them, with the ability to run multiple at once if desired. Since most of the code was repeated between these scripts, it was trivial to combine most of them. For instance, to combine the script that searches for buffer overflows with the script that searches for command injections, the only functionality that needed to be added was to select which set of sink functions would be used.

There are other scripts such as one which finds a path from a source of shared data such as RAM or an environment variable which was slightly more difficult to integrate as it required more changes to the main loop of the program. However, they were successfully integrated along with the rest.

5 Results

GhidraVSA

The GhidraVSA plugin successfully integrates with Ghidra and can be downloaded and installed by anyone who wishes to use it.

GhidraVSA can correctly determine the values of variables in a compiled binary, whether they are stored in a register or in memory. In most simple cases, this works almost flawlessly. For instance, the value of local variable that determines whether to enter a conditional block is correctly identified to contain a value that fits the condition. For another example, the value of a variable that is set to the return value of `atoi("1")` is correctly output as the integer 1.

In more complex cases, the tool is more prone to error. For an example, consider the following code.

```
int a = 0;
int b = 400;
int c = 0;

a = atoi(argv[1]);

if (a > 100) {
  b = a;
}
else {
  c = a;
}

if (b < 300)
  fun(b);
else if (c > 50)
  fun(c);
else
  fun(200);
```

Listing 1: An excerpt from C code used to test GhidraVSA

You will notice that the range of values that can be passed to the function `fun()` is between 51 and 299 inclusive. More specifically, the call to this function can happen at three different places, with the range of possible values split into 101-299, 51-100, or 200.

When GhidraVSA was tasked with finding the possible values of argument to `fun()`, it noted a recursion depth error during the construction of the VFG. However, it correctly recognized two of the three states. It successfully returned the possible input of 200, as well as the range 101-299, but it was missing the range from 51-100.

SaTC

Using my updated version of SaTC, I analyzed 35 firmware images, some through all stages of the SaTC pipeline and some only partially. The filesystem for each firmware was extracted with Binwalk in preparation for analysis.

The improvements to SaTC's modularity facilitated my analysis. In the cases where the first two stages of the pipeline identified several border binaries as containing keywords of interest, the isolation of the third stage of the pipeline allowed me to run it only on the binaries selected.

When running the second stage of the pipeline, some binaries caused the script to crash as Ghidra was unable to read them. This could be due to several reasons, but the most likely is that they were corrupted during extraction.

The sink functions added to the implementation of SaTC resulted in many additional potential taints to be found during the second stage of the pipeline. The addition of the sink function `popen` alone resulted in over 600,000 new potential taint paths to be discovered. The number of these potential taints that present real vulnerabilities was unable to be verified due to limitations in the third stage of the pipeline. More detail is provided regarding these limitations later in this section.

There was much repetition in the border binaries identified by the first two stages of the pipeline. The distribution is shown in Table 1.

Binary	Count
`cfg_manager`	19
`boa`	16
`tr69`	12
`logic`	2
`rc`	2
`lighttpd`	2
`webupg`	1
`nvram`	1
`image2d`	1
`datalib`	1
`net-cgi`	1
`wl`	1
`hostapd`	1
`wpa_supplicant`	1
`tcpdump`	1
`ubntbox`	1
`gpg`	1
`pppd`	1

Table 1: Counts of each border binary found 65 firmware images

Of the binaries that were found, `cfg_manager`, `boa`, and `tr69` were found far many more times. Of these three, the sizes of the `tr69` and `cfg_manager` binaries were much larger than `boa`. Additionally, `boa` was found to be associated with an application called Boa Webserver, a lightweight web application built for embedded devices that was last updated in the year 2005. For these reasons, this binary was focused on for further testing.

Unfortunately, the taint check step of the pipeline failed for every instance on which it was tried. The most common cause of failure was that the taint check ran indefinitely until memory resources were exhausted. In some cases, it terminated but no taint was found.

Several different instances of the `boa` binary were tested. They ranged in size, perhaps due to their compilation options or architecture, as well as in number of potential taint paths found by the previous pipeline, all with the same result. I also compiled the application from its publicly available source code for both the x86 and ARM architectures and these also failed.

The taint check step was run for several other border binaries, but none were successful. Most were also due to the reasons mentioned earlier, but some failed to run at all, crashing when angr attempted to load the binary. Since Ghidra was able to open all binaries that made it to this point, it is likely not due to an error in the firmware extraction. It is more probably due to an issue with angr itself, which is still in development and prone to issues.

The taint check does not run indefinitely for all binaries. It was verified to successfully find a taint path in several test cases that I developed. However, these test cases are much smaller than the fully developed binaries that exist in the firmware, consisting of only a handful of functions and function calls rather than thousands. While the taint check engine implementation may be correct, it does not perform adequately at scale.

6 Discussion

angr provides the opportunity for valuable tools to be developed, but it has severe limitations. The number of paths grows significantly with the size of the program, which makes it challenging to use at scale. This results in either lengthy compute times, or inaccurate results due to simplifying assumptions made in order to speed up computation.

In GhidraVSA, this issue expresses itself in inaccurate value ranges for the symbolic variables. In SaTC, the taint check engine will fail to terminate within reasonable time, running until memory is exhausted. In some cases it will terminate quickly with no taints found, likely due to simplifying assumptions made.

angr suffers from the known problems that symbolic execution engines face: path explosion and accuracy issues. While these limitations will always be present during symbolic execution, the tools that rely on angr can still be improved to work past the restrictions as much as possible.

In the future, this work can be extended by attempting to run these tools on improved hardware. This would speed up the taint check algorithm and provide more memory resources, potentially allowing its completion. This would be useful for more fully exploring the limitations of the software.

7 References

[1] Chen Libo, et. al. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. https://www.usenix.org/system/files/sec21-chen-libo.pdf, 2021.