Static Taint Analysis with Ghidra

Jacob Gilhaus
Spring 2022 - Master's Project
Department of Computer Science and Engineering
Washington University in St. Louis

***Abstract*—The prevalence of the Internet of Things brings with it an increased risk for potential harm due to the vast number of potentially exploitable vulnerabilities on the web. This paper proposes an improved static taint analysis engine that can identify vulnerabilities in IoT device firmware. The script to achieve this improvement places further emphasis on Ghidra as a tool for performing taint analysis, identifying generalizable vulnerabilities such as buffer overflows and command injection as well as firmware control-flow related vulnerabilities like a firmware downgrade attack. The usage of the Ghidra API and its Basic Block Model allows increased access into the program during static analysis, as well as the ability to recover some semantic information from the binary. By increasing performance and incorporating features like cryptographic function identification and control-flow graphing of firmware binaries, this engine represents an improvement in the design, performance, and utility of past Ghidra taint analysis engines.**

I. Introduction

The Internet of Things (IoT) and the mainstream embrace of smart devices has created an explosion in the sheer number of devices and network connections in the digital world. Each and every device has some form of firmware to support its operation, and inherent in the influx of IoT devices is an emphasis on low-cost materials and accelerated development. The low-cost and ease of replacement for these devices often means less secure firmware and provides an access point for malicious users looking to take advantage of these new machines on the network. To combat the growing botnets and the challenges of so many IoT devices, recent research has placed an emphasis on detecting bugs in IoT firmware.

One such strategy use is the implementation of taint analysis engines to reveal bugs in the firmware images through static analysis. This strategy requires the use of reverse engineering tools to analyze a firmware image and the binaries within it. Ghidra, with its relatively recent public release by the NSA in 2019, has become a key analysis tool in many previous firmware analysis works and provides a great tool for taint analysis. By leveraging the Ghidra API [1] and its extended functionality over many other reverse engineering tools, we can build an effective taint analysis engine for analyzing binaries such as firmware images, finding bugs that might otherwise lead to further exploitation in the wild.

This work expands on the ideas shown in the Sharing More and Checking Less (SaTC) paper [2], which leveraged shared keywords between frontend and backend binaries, and applies them to firmware by enumerating the possible paths to the sink function. The buffer overflow and command injection vulnerabilities trace from a source keyword, while the control-flow analysis portion of the script finds the main function as its source. SaTC additionally identifies the possibility of inter-process communication (IPC), which this work also takes into account.

The engine developed in this project improved off of previous efforts by utilizing the Ghidra Basic Block Model to construct control-flow graphs. Additionally, this script also extracted data from the symbol table and identified cryptographic functionality along vulnerable paths. A useful example for this application is in the case of a firmware update, since updates should perform operations such as checksums and version checking to avoid malformed update data and firmware downgrade attacks. This allowed us to explore a taint engine for multiple classes of vulnerabilities, with accurate control-flow, detection of relevant operations on the flow path, and multi-binary compatibility.

II. Background

The research in this project required some additional knowledge and preparation. The relevant background information is best broken down into firmware security, Ghidra, and taint analysis, though other related topics are also discussed throughout the paper without the need for extensive discussion here.

A. Firmware Security

Firmware is a key piece of software written into devices to support interaction with hardware and other basic functionality. Due to the integral nature of a device's firmware, it is vital for correct functionality of the device but also presents its own set of unique challenges. Especially in the world of IoT devices, firmware is essential to establishing connectivity and enabling the smart operations the device can support. There are two flavors of firmware, one that supports a file system and one that does not. The latter is called a bare-metal firmware image, though this engine focuses on the former.

In terms of security, firmware needs to receive software updates in order to patch security vulnerabilities and provide new functionality as it becomes necessary. In solving some security problems via firmware update, the device also opens the door to potential vulnerabilities. Part of the inspiration for this research is to help detect bugs in firmware, including those introduced in the update sequence.

B. Ghidra

Ghidra is an open-source reverse engineering tool developed by the NSA. Released to the public in 2019, the tool is a direct competitor to IDA PRO, OllyDbg, and Binary Ninja. Written in Java and C++, Ghidra also supplies an extensive and useful API. The API supports Java as well as Python scripts through the Jython interpreter. By foregoing the GUI, users can run Ghidra headless followed by custom postscripts that can further analyze the binary. The headless approach is leveraged in this research to statically run Python taint analysis scripts.

C. Taint Analysis

Taint analysis or taint checking is a security process centered on the presumption that any user supplied input could lead to undesired consequences when the user supplies something malicious or unexpected. One clear example is a buffer overflow vulnerability. Given normal user input, the vulnerability might never be discovered, but taint analysis checks the impact of this data as it travels through the program. By marking other variables and function calls that are dependant on this input, taint analysis can reveal reliance on user input by known vulnerable functions.

In some cases, taint analysis can be closely related to control-flow. For example, we may still want the full control-flow graph of the binary, so the script has the capability to start at the program entry point as opposed to the location of the keyword. This portion of the script differs from a more general control-flow graphing program in that it still specifies vulnerable sink functions despite nonspecific sources.

III. Related Work

Before delving into research details and the Ghidra engine, this project and the associated research drew on the legwork of others in the fields of reverse engineering and firmware security.

First, many others have made attempts at writing effective taint analysis scripts for reverse engineering tools and developed a useful framework for thinking about and solving such problems. Binary Ninja did so using the vulnerable function `memcpy` [3]. Shortly after, River Loop Security capitalized on the release of Ghidra to produce a similar taint analysis engine that works backwards from `malloc` calls [4]. Their blog post outlining P-code was of much conceptual use [5]. Additionally, FirmXRay [6] introduced the firmware address space through bare-metal binaries and the prospect of automatically detecting the entry address, though this is unnecessary for file system images. An Attify post [7] also drove this possibility home with an example using Ghidra, further validating the efficacy of Ghidra.

Notably for firmware security, RFC 9019 [8] from the IETF was quite useful in understanding the firmware update process. The RFC focused on the architecture of the process and the necessity for standardizing a firmware update server and tracking server for added security through firmware updates, but it was still useful for understanding the motivation and mechanisms for firmware updates.

Finally, Sharing More and Checking Less (SaTC) and Karonte [9] were most closely related to our firmware security research and the Ghidra scripting specifically. Karonte provided a taint analysis engine for IPC and multi-binary interactions that SaTC used in their own work. SaTC built off Karonte by introducing the shared keyword concept. Most effective in routers and web cameras, by far the most popular IoT devices, the shared keyword idea identifies strings in frontend binaries like a router's web interface. Then, it searches backend binaries for the same keyword and performs taint analysis on any found backend keywords in border binaries. Simply put, border binaries accept input such as user requests over the network. For example, the border binary `httpd` is common in our work since it runs an Apache server.

The shared keyword intuition is that the identical naming of keywords represents a dependence relationship between the frontend and backend binaries, and was shown to be successful at identifying bugs in the SaTC analysis. SaTC introduced Ghidra scripting as a mechanism for identifying plausible paths from sources to sinks using assembly-instruction-level analysis using a callmap, but then went on to also utilize ANGR, a symbolic execution engine. Instead of deferring to ANGR just yet, this work explores a redesign of Ghidra scripts in the shared keyword context to explore the potential of the Ghidra API for use in taint analysis.

IV. Design

The design goal of this project was to use the inspiration of past taint analysis research using Ghidra scripting to create a script that could identify firmware bugs. The threat model is that of a malicious attacker attempting to jeopardize the firmware, including through the update process, which typically takes place over a network connection to an update server. The attacker is assumed to have access such that they can reach the device over the network. Some potential attacks include buffer overflows, command injection, and uploading malicious firmware as attacks on the update mechanism.

In detecting buffer overflows and command injection, we use taint analysis to find user influence on vulnerable sink functions. For update attacks, the firmware update code built into the device should check correct transmission via a checksum and utilize an authentication verification method to prevent bricking the device or updating to a malicious image, as examples. Additionally, if the update code does not check the version of the received firmware it may be at risk of a firmware downgrade attack. This kind of attack may result in reintroducing a previously patched vulnerability the attacker can then exploit.

In incorporating some of the key concepts from SaTC and others into the firmware update model, there were some design decisions that needed reconsideration under the new context:

- Level of detail
- Level of operation
- Exploit specific features

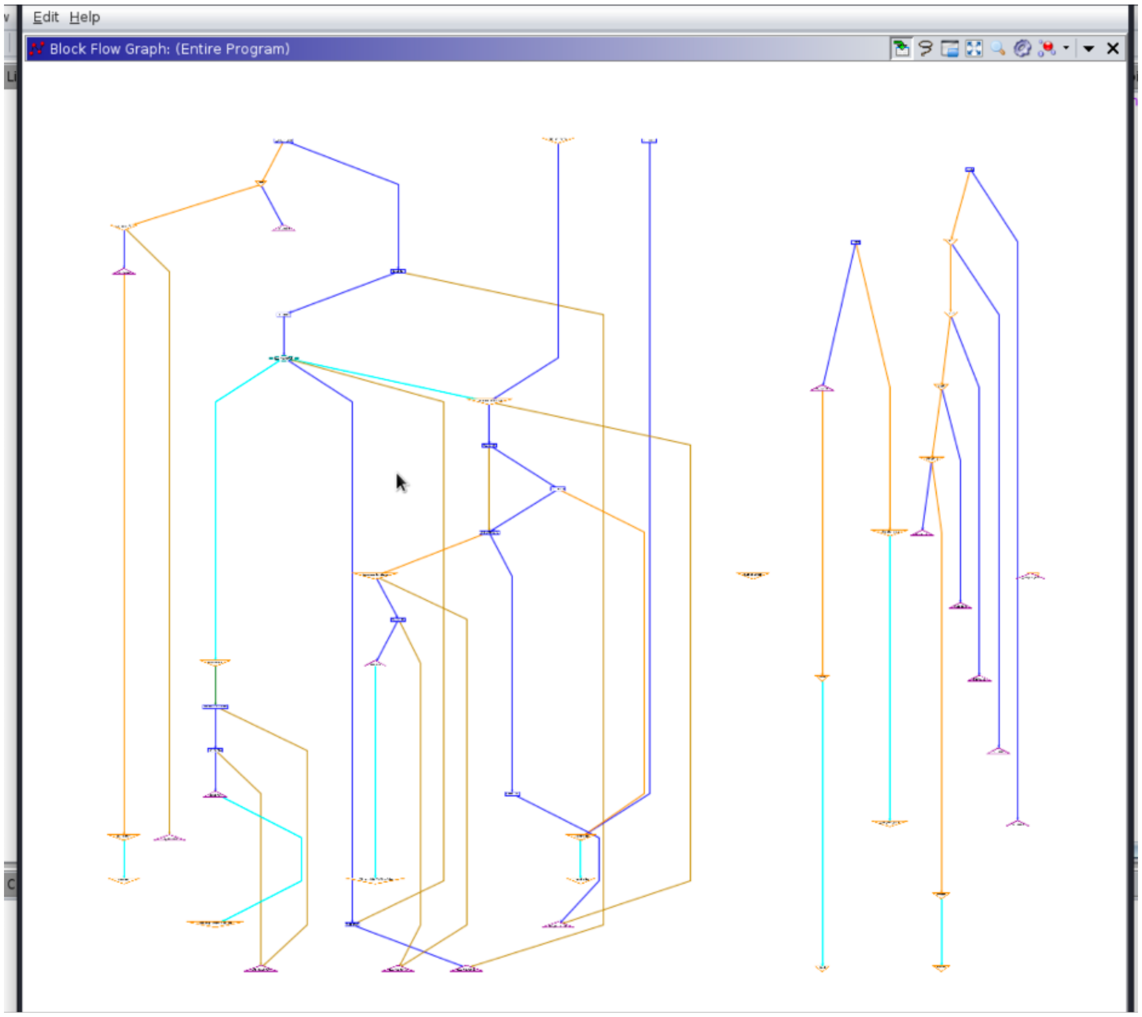

Fig. 1. An image of a Ghidra's block flow graph for a simple nested function call to system. The left branch of the far right tree shows these calls. The code is depicted in Listing 1. This kind of visual representation may be familiar for those with experience in IDA PRO.

Each of these differences are assessed individually. In this project the chosen design was fine-grained, at the block-level, and included multiple vulnerability types as well as maintaining multi-binary compatibility.

A. *Fine-grained*

First, the coarse-grained nature of the scripts was insufficient. The SaTC approach simply walked through the entire program and built a callmap mapping each function call to its referencing addresses. Then, sorting the callmap by address it simply performed a depth-first search looking for calls to sink functions. This approach doesn't take into account the control-flow of the program and lacks detail in terms of tracking and counting all vulnerable paths from the program entry point.

To build on this idea, the renovated script is able to use a Ghidra API model to perform its own depth-first search strategy, but it can also collect and enumerate all paths to the sink in the binary. This is valuable since in some cases, such as the update process, a flow **should** exist to the sink function, but we would like to identify the particular flows that ignore important checks because these paths are the source of vulnerability. This is in contrast to simply detecting the plausible existence of one or more paths.

B. *Block-level*

Furthermore, most past Ghidra work operated at the instruction-level. While this can be effective, it is unnecessary to iterate at this level and even loses some semantic information. For example, in building the callmap the SaTC `search()` function iterates through the entire program one instruction at a time until it reaches the end, constantly looking for callees despite most instructions not having any. Luckily, the Ghidra API provides us a more useful model, the Basic Block Model (BBM). The BBM is similar to the control-flow diagrams generated in the Ghidra GUI as seen in Fig. 1, and the model groups together consecutive instructions while enforcing that any potential branch instructions are always the final instruction in a block.

Thus, by leveraging the BBM we can not only skip over instructions, but we can also follow the control-flow more precisely with API calls like `block.getDestinations()`, which will return all Basic Blocks the current block can flow to. This is analogous to the edges flowing down the graph to other blocks in the GUI representation. BBM also allows us other useful API calls such as `block.getFlowType()` which returns the flow to reach the destination block, which includes conditional call, unconditional jump, terminator, fall-through, and more.

C. Exploit Oriented and Multi-binary Compatible

Finally, this design emphasized multiple exploit-related vulnerabilities as opposed to one at a time. To do so, the design increased the functionality of the Ghidra scripts to record additional information about the program and the vulnerable paths. Buffer overflows, for example, can simply track taint sources and find a path to a vulnerable sink function such as `strcpy`. Command injection works similarly, though with different sink functions. In an update vulnerability scenario, we construct a control-flow graph and track relevant information along the path to sink functions like `system("reboot")` that will initiate the reboot process.

Again, the Ghidra API supplies useful methods that allow us to retrieve both symbol table and function information. Once Ghidra has collected this information about our program, we can check the reference addresses of symbols along the path as well as function calls. If the naming of such symbols or function calls can be recognized as containing cryptographic or version checking information, such as `calculate_checksum`, we can record the existence of this symbol or function in a detected path.

Another important capability of this script is that it was designed with the possibility of IPC in mind. Sometimes a firmware image will enlist separate binaries to perform complementary tasks. In such cases it is helpful to detect inter-process communication, which often uses environment variables or shared memory, and run the taint analysis script on multiple binaries. Consider an example where the firmware update binary simply checks the environment variable set by the version checking binary. The firmware update binary may not utilize any checksum or cryptographic functions, but the checksum is still verified. Thus, the script should gracefully handle the case in which this occurs. We should also be able to track user inputs passed between binaries for buffer overflow and command injection cases.

The way the script handles this is by leveraging the work of another Ghidra script, though they could potentially be combined into a single workflow in the future. This other script detects IPC and writes to a file detailing the binaries that communicate over environment variables, shared memory, or their own shared files. Then, this script can read from that file and take that relationship, and the specified addresses in which the communication takes place, into account by parsing the cases where IPC refers to this binary and using the reference address as a source.

V. Implementation

The implementation of these scripts took place in the private GitHub repository for the firmware update research. The repository was cloned and a branch was created dedicated to this work on Ghidra scripting and taint analysis. For script development, a shared folder with repository access was used on a VMware virtual machine running Kali Linux. The Ghidra setup process is outlined in detail on the Ghidra Installation Guide [10], though the basic requirements are supporting JDK and setting up the decompiler to support your specific system.

VI. Evaluation

To verify correctness, debugging and verbosity were utilized during the development process as well as extensive use of the API documentation. Various examples stressing different points of emphasis in the script were also developed as test binaries such as those depicted in Listings 1 and 2. In the shown examples, emphasis was placed on testing the control-flow graph since this addition is novel. Each test binary suggests accurate execution of the scripts. These test binaries include a nested function call binary shown in Listing 1, a multiple path binary shown in Listing 2, a similar multiple path binary that would result in only 1 path being possible, a binary with a max calculation and library calls such as `printf`, and a few combinations of these approaches. Other test binaries include those that simply pass user input to a `strcpy` or `memcpy` function to cover the full landscape of attacks for this engine.

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

int callSystem(){
    system("ls");
}

int nest3(){
    callSystem();
}

int nest2(){
    nest3();
}

int nest1() {
    nest2();
}

int main() {
    nest1();

    return 0;
}
```

Listing 1. An example program that was developed for testing. The program uses nested function calls and eventually calls system as the sink function. The block graph for this program is shown in Fig. 1.

Unfortunately, there were no benchmarks for this kind of taint analysis known of at the time of development to fully verify the scripts. However, a manual review of some of the smaller binaries from the SaTC dataset also supports the script correctness. The images in the dataset could be unpacked using binwalk since they included file systems. Given the file system produced by binwalk, tools developed by the rest of the firmware team extracted the files of interest for taint analysis. Some of the tested images included the Netgear R6200v2 and D-Link 823G routers.

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

int callSystem1(){
    system("ls");
}

int callSystem2(){
    system("pwd");
}

int nest2(){
    int r = rand() % 10;

    if(r <= 5){
        callSystem1();
    }
    else{
        callSystem2();
    }
}

int nest1() {
    nest2();
}

int main() {
    nest1();

    return 0;
}
```

Listing 2. Another example program that can take multiple paths to system (the sink function).

`httpd`	**BufO**	**CmdI**	**Control-flow**
BBM	0.362 (5)	0.331 (3)	6.475 (27/39)
SaTC	4.946 (4)	2.499 (1)	X

Table 1: `httpd` timing measurements in seconds comparing the Basic Block Model to SaTC's command injection script. Binary is from the Netgear R6200v2 firmware image. Parenthesized is the number of detected paths from source to sink. For control flow the number of vulnerable paths and total number of paths are parenthesized.

`check_fw`	**BufO**	**CmdI**	**Control-flow**
BBM	0.006 (0)	0.003 (0)	0.276 (4/4)
SaTC	0.018 (0)	0.014 (0)	X

Table 2: `check_fw` timing measurements comparison in seconds. Also from the Netgear R6200v2 firmware image.

Performance-wise, an identical timing mechanism was used as in SaTC to measure the time the taint analysis took. The findings suggest comparable or improved timing results from SaTC to this taint analysis technique. The overall overhead was reduced by swapping the instruction-level for block-level analysis, but constructing the control-flow for the entire binary expanded the footprint of the taint analysis. Additionally, the BBM taint analysis technique trades off searching for additional keywords heuristically for a more detailed analysis of the cryptographic and version checking mechanisms in place. Altogether, maintaining similar combined timing measurements between these two - now quite different - taint analysis techniques is a surprisingly positive achievement. Not only does SaTC require separate scripts for each vulnerability, but the BBM engine performs full control-flow. Tables 1 and 2 detail timing measurements in seconds for the Basic Block Model and the SaTC scripts. They are all run on the Netgear R6200v2 firmware image with the `httpd` and `check_fw` binaries.

Notably, the depth-first search with the Basic Block Model found additional paths thanks to its increased efficiency and depth, and did not result in a performance blowup when comparing both small and large binaries. In fact, comparing solely buffer overflow and command injection timing, BBM can perform nearly 10x better. BBM proves more effective for the shared keyword checks for buffer overflow and command injection, and full control-flow performs comparably to SaTC. This indicates it would be well within the acceptable performance window, and even with the expanded footprint these results highlight the relative speed of the BBM engine.

Lastly, for `httpd` the script is able to identify cryptographic functions on 12 of the 39 paths, indicating the other 27 are potentially vulnerable to update-related attacks like firmware downgrades. There are potentially excessive false positives as most paths still must be analyzed, but the path reduction does show a proof of concept for constraining the scope of control-flow path analysis.

VII. Discussion

While this script represents progress, challenges also present themselves. First, the entry point variation caused inaccuracy in certain binaries. When not starting at the keyword address, finding the main function was typically simple and effective, but sometimes main cannot be found. Without a clear entry point the script compromises by starting iteration at the minimum program address, though the resulting control-flow is altered as a result. Also, recovering certain semantic information from branches, such as identifying the specific comparison done, presents a challenge due to the inherent uncertainty in the reverse engineering process, though some constant arguments or strings can be detected on a function by function basis. The River Loop Security approach that utilizes backtracking at the P-code level could provide some deeper analysis, though it complicates the control-flow graphing process. Even simply identifying sink arguments reliably would be a marked improvement in using Ghidra for taint analysis.

Another relevant issue is the script verification process. The cryptographic function identification proved successful for some common operations, but the miss rate of this technique is not yet clear. This results in unnecessary path analysis and while manual checking is possible, an automated solution would be far more preferable. This was a potential inspiration for integrating symbolic execution and ANGR into SaTC and allowing it to do more heavy lifting. While these limitations certainly provide roadblocks, they are potentially surmountable as future work proceeds on reverse engineering tools and similar scripts in well-defined scenarios.

VIII. Conclusion

Here a novel strategy for performing static taint analysis with Ghidra is proposed. This technique offers detection of multiple kinds of vulnerabilities, performs control-flow analysis, and identifies the existence, or lack thereof, of relevant functions along potentially vulnerable paths. The additional overhead added to the Ghidra taint analysis engine achieves comparable performance, but the added depth benefit, increased control-flow insight, and nearly 10x execution times for buffer overflows and command injection make up for this. The development of such a script marks a design, performance, and utility improvement over past Ghidra taint analysis engines.

IX. Acknowledgements

This project would not have been possible without the support, assistance, and availability of the entirety of Dr. Zhang's Computer Security and Privacy Laboratory. Of exceptional note were: my advisor Dr. Ning Zhang, Yuhao Wu, Jinwen Wang, Jack Zhai, and Benjamin Gilman. I worked closely with each of these individuals at one point or another and greatly appreciate their support as well as the hospitality of the entire lab.

References

[1] National Security Agency. Ghidra API Documentation. https://ghidra.re/ghidra_docs/api/index.html

[2] Libo Chen et al. "Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems." USENIX Security Symposium, 2021.

[3] Sophia d'Antoine, Peter LaFosse, and Rusty Wagner. "Be a Binary Rockstar." Infiltrate 2017. https://vimeo.com/335158460

[4] Alexei Bulazel and Jeremy Blackthorne. "Three Heads are Better Than One: Mastering Ghidra." Infiltrate 2019. https://vimeo.com/335158460

[5] Alexei Bulazel. "Working with Ghidra's P-code to identify vulnerable function calls." https://www.riverloopsecurity.com/blog/2019/05/pcode/

[6] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. "FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware." CCS 2020.

[7] Attify user Barun. "Analyzing bare metal firmware binaries in Ghidra." https://blog.attify.com/analyzing-bare-metal-firmware-binaries-in-ghidra/

[8] B. Moran, H. Tschofenig, D. Brown, and M. Meriac. "RFC 9019 A Firmware Update Architecture for Internet of Things." Internet Engineering Task Force Request for Comments, April 2021.

[9] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware." In Proceedings of the 41st IEEE Symposium on Security and Privacy, 2020.

[10] National Security Agency. Ghidra Installation Guide. https://ghidra-sre.org/InstallationGuide.html