SOLOMON: Static Multi-binary Bug Finding on Embedded Firmware

What does Multi-binary mean?

- firmware typically made up of several hundred binaries
  - 254 on average in their dataset.

Why need multi-binary analysis?

- 1. can reduce false-positives, because some attack will not happen at all.

> Page.1: Thus, any analysis on the receiving binary in isolation would identify bugs that are actually impossible to trigger in practice, and require far more computational resources, as it would have to assume that every input is unconstrained.

e.g. function A in binary X may seem vulnerable, it doesn't check the length of data, but the truth is, function A is only called by function B in another binary Y, and function B will check the length for it before it pass the data to function A. If you check binary X seperately, you will find function A is vulnerable, but if you check binary X and Y at the same time, you will find the data in function A is guaranteed to be correct.

Or, the data is from a read-only file, the attacker has no way to modify the read-only file, so this function is secure even if it doesn't check the length of the data.

> Page2: Thus, any technique that considers all of them in isolation is likely produce a substantial amount of false positives. This is because without considering the binary interactions, each input would need to be treated an unbounded input, when, in reality, many of the inputs are heavily constrained by the binary that produces the data. Moreover, not all of these binaries handle data that is controllable by a remote attacker.

My thought: I think they are not actually false-positives, because the existence of such functions are already a potential threat to the program. Let's say that even if the program have a lot of vulnerable functions which will never be called, the hacker can still try to jump to these functions (e.g. they hijacked the program by using another function, but the buffer length of this function is not enough for a complete shellcode, then they can jump to these "useless" vulnerable functions and inject the shellcode there), and they will also provide ROP gadgets, unless the compiler optimizes these functions away.

- 2. find more cross-binary vulnerabilities which can be exploited by hackers

  > Page.1: Nevertheless, cross-binary vulnerabilities are already being exploited by attackers [41], [47], [14], and, necessitating tools that are capable of analyzing these interactions by tracking user data across binaries at scale.

Do we have multi-binary analysis now?

> Page.2: The requirement for cross-binary program analysis is a major hindrance for current techniques. Most recent work in dynamic and static program analysis focused on the analysis of a single program or module at a time [55], [36], [46]. While some work has focused on emulation-based firmware analysis, no approach exists that can emulate a firmware sample to the extent of interacting reliably with a dynamic analysis engine (success rates range from 13% [10] to 21% [4]). On the other hand, system-wide static analysis techniques are currently only feasible on small-scale toy examples or require source code, neither of which are realistic assumptions when analyzing real-world embedded firmware [12].

What's our contribution?

> Page2: We have a technique allows us to propagate constraints across binaries, which greatly reduces our false positive rate and increases our efficiency. Similarly, our technique automatically identifies the functions which send and receiver user data in each binary, which ensures that our analysis is only performed on binaries which handle externally input data, again reducing false positives and increasing our scalability.

- 1. detecting where user input is introduced into the system, identifying potential interactions between the various components.
- 2. These interactions are then used to track data flows between components and perform cross-binary taint analysis.

- 3. Finally, this taint analysis is used to detect unsafe uses of the user input, which can lead to vulnerabilities.

How to do we do multi-binary analysis?

- 1. Through data dependence
  - i. string-based search, e.g. in XML file
- 2. Through the way they pass data
  - i. functions to share data in different binaries, e.g. setenv/getenv, write/read or, generally, ad-hoc set/get functions

My thought: string-based search cannot fight against obfuscation.

Design

Currently, the design focuses on interaction with remote server, because most firmwares highly rely on network service, because they need to upgrade the firmware remotely.

- 1. Input = unpacked image,
  - usually unpacked using `binwalk`
- 2. Network service discovery:
  - > Page.4: by scoring each binary present in the firmware sample based on the semantics of its functions.
- 3. Binary Dependency Graph (BDG) Recovery
  - identify the binaries produce and consume data
- 4. Static Taint Analysis
  - performs static taint tracking over the binaries in the BDG.
  - to propagate taint over the BDG into other binaries allows us to extract interaction models that describe the flow of attacker- controlled data across binaries
- 5. Security Violation Detection
  - analyzes data reaching taint sinks in the binaries of the BDG to identify potential security issues caused by inter-binary or intra-binary attacker-controlled data flow, which are then reported to an analyst for further inspection.

How to find Network Service

Give every function a parsing score, and filt them. Basically, it is based on four elements:

1. the number of basic blocks, #bb
2. the number of branch it takes, #br
3. the number of memory, #cmp
4. network marks, based on strings (e.g. “remote” or “http”), #net

The score is calculated by a weighted equation:

$$(k_{bb} * \#bb + k_{br} * \#br + k_{cmp} * \#cmp) * \#net$$

Using BINARY DEPENDENCY GRAPH (BDG) to represent the relationship between binaries

Prepare: some key strings, some binaries.

1. Extract all the key strings contained in every binary,
2. For every binary, build its call graph and retrieve all the strings in each binary’s data segement that contain key strings.
3. Find the functions which reference these key strings
4. For each function, examine the paths which can lead to this function, from the directly connected function in the call graph.
5. Detect if the tainted data is sent in these functions.

How to define a typical inter-process communication

1. Through function name (e.g. set, insert…) in the library (e.g. Linux library)
2. Through Socket, relies on symbols that indicate socket operations (e.g. send, recv…)
3. Through file, considers the name of the file
4. Through semantic, e.g. shared memory

Improvement with current Taint Engine

1. reduce false positive results

- The taint is not propagated for unfollowed function calls.
- taint all of the function’s arguments and its return value when any of the function’s arguments are tainted and the function is not explored.

2. be able to write an ad-hoc untaint policy

- a symbolic expression is untainted when all of its tainted symbolic variables become untainted.