Report on Saluki: Finding Taint-style Vulnerabilities with Static Property Checking

Jacek Janczura
404975, Software Engineering of Embedded Systems, Technische Universitaet Berlin,
janczura.jacek@gmail.com

Abstract

This report explains the architecture and Saluki's idea for checking taint-style (data dependent) security properties in a binary code. Saluki is a new tool which is capable of finding a large number of CWS[1] vulnerabilities in real programs. Saluki uses a mixture of static and dynamic taint analysis to follow data dependent facts. Saluki is proved to be capable of finding vulnerabilities in COTS[2] including 0-days.

I. INTRODUCTION

Recently vendors continue to ship vulnerable programs. Unfortunately to protect their ”know how” and intellectual property, source code is not shipped, which is a big issue. Modern compilers and run-time libraries have introduced significant complexities to a binary code, which negatively affect the capabilities of binary analysis tool kits to analyze binary code. [3] To cope with analysing modern binary files, a data flow analysis is used for an accurate detection of a wide range of attacks on a shipped software, including those based on memory corruption, format-string bugs, command or SQL injection, cross-site scripting, and so on.([4], [5])

An example of data flow analysis used for finding such vulnerabilities is a taint analysis. The main goal of taint analysis is to identify data flows from attacker-controlled sources to security-sensitive sinks that do not undergo sanitization.[6] [7] There are two types of taint analysis, dynamic and static. [8] Dynamic requires run time support and can achieve low coverage, both of which can result in missed vulnerabilities. Static analysis, promises to reason about entire functions or whole programs at once by abstracting program state. As a result, static analysis tends to miss fewer problems, but can suffer high false positives if not done with care and is extremely time consuming. [9]

The solution for that problems is Saluki. This new tool, introduced by a research group from the Carnegie Mellon University, proposes the third approach - mixed taint analysis based on random execution. This type of data flow taint analysis is much faster than the conventional static approach and finds much more vulnerabilities than dynamic analysis.[9]

In this report after after introducing some basic concepts, we will focus on the Saluki's architecture, steps of its execution, some real life results and tests conducted using vulnerable binaries.

[1]CWS- Common Weakness Enumeration - list of software weakness types [1]

[2]COTS - Commercial off-the-shelf - products are packaged solutions which are then adapted to satisfy the needs of the purchasing organization [2]

II. Control Flow Graph - CFG

A CFG is defined to be a directed graph, consisting of vertices that represent basic blocks of the code and edges that represent control flow, $G = (V, E, V_e, V_x, T)$, where: ([10], [3])

- $V = B\{v \perp\}$ is a set of nodes corresponding to all basic blocks B and a special sink node $v \perp$ that has no instructions or outgoing edges;
- $E \subseteq V \times V$ is a set of control flow edges between nodes;
- $V_e \subseteq V$ is a set of entry nodes;
- $V_x \subseteq V$ is a set of exit nodes;
- $T : E \rightarrow \{intraprocedural, interprocedural\}$ assigns a label to an edge.

The basic blocks B are defined in a conventional way. Each basic block $b =< i_0, i_1, ..., i_n >$ is a consecutive instruction sequence with i_0 being the only entry and in being the only exit. The sink node $v \perp$ is used to represent unknown control flows, mainly caused by indirect jumps and indirect calls.[11] An example of such graph is depicted in a Fig. 1.

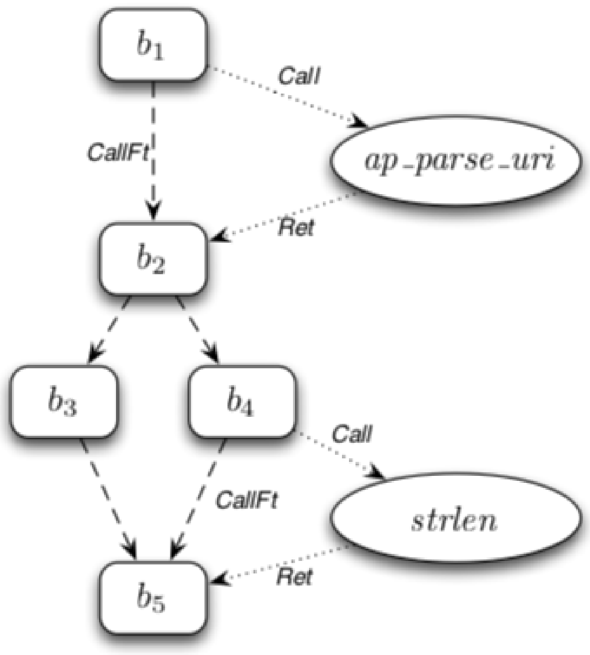

Fig. 1: Sample Control Flow Graph.
Dashed arrows represent intraprocedural edges, while dotted arrows represent interprocedural edges. Functions are summarized as ellipses (f) [10]

III. Taint-style Vulnerabilities

The term taint-style vulnerabilities has its roots in taint analysis, a technique for tracing propagation of data through a program.[7] Taint analysis is based on the observation that in order for an attacker to change the execution of a program illegitimately, attacker must cause a value that is normally derived from a trusted source to instead be derived from his own input. We refer to data that originates or is derived arithmetically from an untrusted input as being tainted. [12]

Taint analysis can be used for example to check if the data received to a socket by the function $receive()$ and saved in some buffer does not leak. It means that it is not sent away by the function $send()$.
In an example shown in a Fig. 2, functions and buffers that cannot be dependent one to another need to be specified. Any data saved by $receive(*buf_a)$ can not be sent away by $send(*buf_b)$.

Taint analysis is divided into three steps:

- **Seeding** - during seeding all the $receive()$ functions need to be find in the CFG. Than the memory cells, where the received data is saved, need to be tainted. Each tainted memory cell has its index number for tracing the leakage path and taint flag. Taint flag informs if that specific memory cell has been already tainted or not.

- **Propagation** - taint propagation resembles spreading of a virus. In propagation step all the paths in CFG from $receive(*buf_a)$ to $send(*buf_b)$ are traversed and each used memory cell is tainted.

- **Checking** - during checking each one of the memory cells in buf_b is examined for the taint. Presence of the taint in buf_b means that received data may be leaked away.

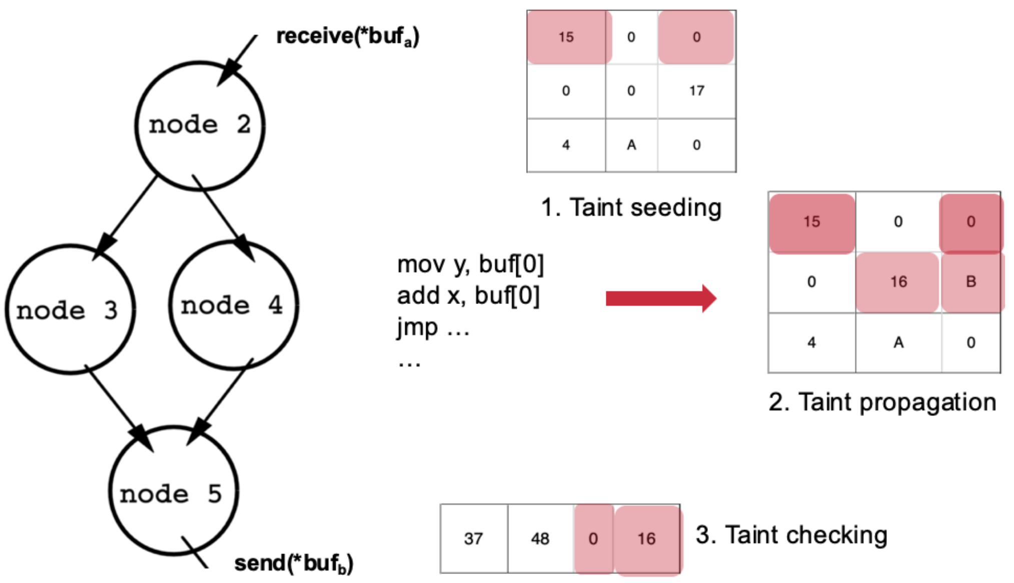

Fig. 2: Visualisation of taint analysis

A. Dynamic taint-style analysis

In dynamic taint analysis, we first mark input data from un-trusted sources tainted, then monitor program execution to track how the tainted attribute propagates and to check when tainted data is used in dangerous ways. [12] Since dynamic analysis is done in a run time, by running the binary and following execution taint, for each different input whole program needs to be rerun from the beginning. In consequence to check all the possible states thus cover all the possible bugs the computational complexity of such approach will be extremely high and a scalability of it will be really poor. That is why in dynamic analysis only subset of the possible inputs is used to emulate working program. Therefore taint analysis is an example of an under approximation Fig.3 and can not cover all of the possible vulnerabilities. [7][4]

- Pros:
  - It is fast
  - No false positive
- Cons:
  - Detects only bugs triggered by an executed path at the run time
  - Always starts at the entry point

B. *Static taint-style analysis*

Although static analysis examines all code paths, it has weakness of a high execution time. Due to it's content independency, static analysis is possible to examine dead code that actually can not be executed in a run time. Consequently static taint-style analysis is an example of over approximation Fig.3. Due to the large number of an external unreachable by the program in the run time states, in the static analysis we observe a large number of false positives. On the other hand all the states reachable by the program are the subset of the states reached by static analysis. It means that among many false positives, all of the possible bugs will be caught. [6][7]

- Pros:
  - Detects all the bugs
- Cons:
  - Many false positives
  - Takes a long time

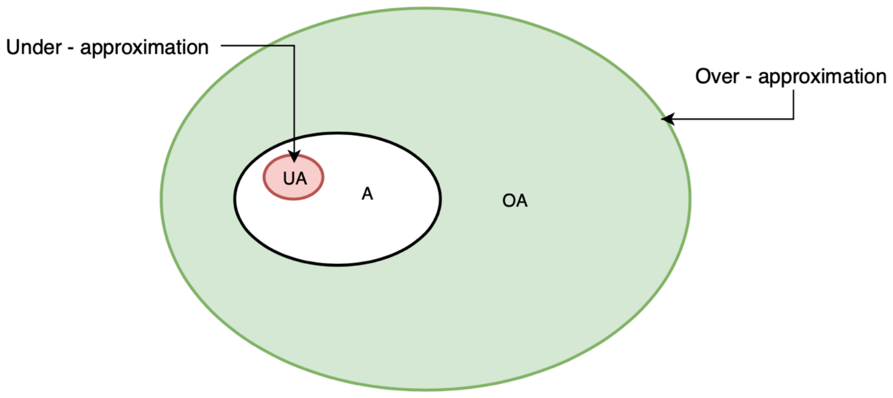

Fig. 3: Visualisation of over and under approximation.
A - a set of all the possible states of a binary, UA - an under approximation of A, OA - an over approximation of A [13]

IV. Saluki Architecture

Saluki creators came up with a new and novel approach. Instead of under approximating the states in a run time or statically slowly over approximating all the possible inputs and impossible states, in case of Saluki - random parts of code are executed.

In a seeding part all instructions and policies that a user wants to check are being found in a control flow graph and tainted. Then in a propagation part, plugin called μflux executes random parts of the CFG supplying the random input and goes down the graph.

Thanks to μflux and executing random parts of CFG, Saluki does not need to start every time from the beginning as it is in dynamic analysis. Additionally randomisation of an execution point, lets Saluki go as far as it is statistically possible in CFG, where in dynamic analysis going that deep entail extreme rise in the computational complexity.

On the other hand randomisation of the input makes Saluki achieve a normally unreachable states like in over approximation. That is why the states approximation of Saluki is the mixed one.[14]

Saluki's execution can be divided into five steps. Its overall architecture is shown in a Fig. 4. [9] :

A Load in the specification.
B Parse the binary into an intermediate representation (IR[3]) suitable for analysis.
C Run μflux[4] to collect data flow facts about executions from every specified source.
D Run a solver over the policies, program, and collected facts. The solver determines whether the property holds or not.
E Saluki outputs example paths where the property does not hold. The actual output is not a full path, but instead a condensed form of the tainted instructions and a flow ID.

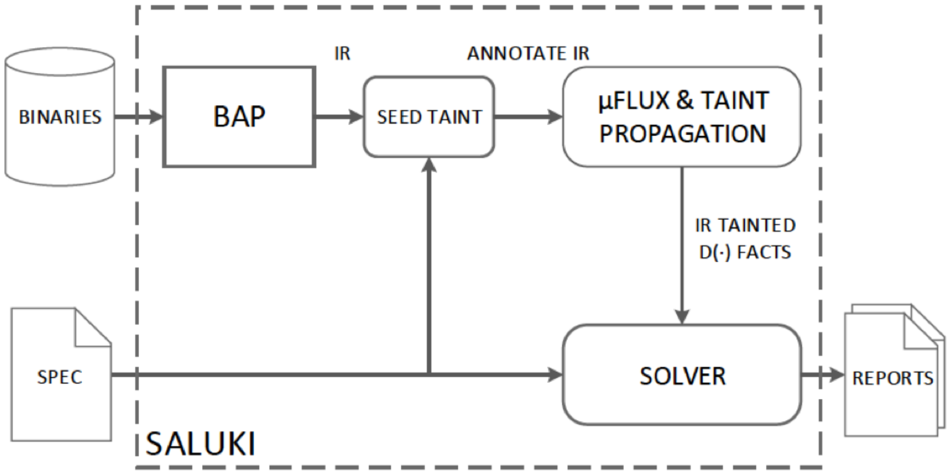

Fig. 4: Saluki Architecture [9]

Specification: The first step of running Saluki is loading the vulnerabilities specification. Saluki has already implemented a database - CWE[5] with dozens of vulnerabilities.

[3]IR can be simplified for better understanding as creating a control flow graph of whole binary
[4]μflux works as a taint propagation
[5]CWE - community-developed list of common software security weaknesses

Additionally user can load customised, user-defined policies using the Saluki's language.

Listing 1: Command injection example

```
prop recv_to_system ::=
recv(_,*buf,_,_), system(*cmd) |- never
s.t. cmd/buf
```

In the Listing 1, there is shown a policy in Saluki language for finding a command injection.

Command injection vulnerabilities arise when input flows from an input source to a sink function that executes code.

In this case system should never use data from recv (data received from the socket). Otherwise we can assume that a command in a system shell was executed - system used data from the socket.

The only limitation is that Saluki does not specifically reason about memory corruption vulnerabilities such as buffer overflows.

Binary processing: The binary is loaded to the Saluki. Than Saluki runs a BAP[6] (BinaryAnalysisPlatform) an open-source plugin-based binary analysis framework. Deriving a CFG from a binary is a difficult problem on its own right. The design of these algorithms is a challenge due to the presence of both indirect control flow that cannot be statically analyzed and data inter- mixed with code.[10] To that purpose Saluki uses BAP recursive-traversal parser, that follows statically determinable control flow to discover as much code as possible, and makes use of backwards slicing and heuristic techniques to identify the targets of indirect jumps (e.g., jump tables) and functions that are only reached via indirect calls. ([15], [16])

Taint Seeding: Saluki analyzes the specification for variables used in constraints. Each constraint variable is linked to a program location, which is then marked in the IR as a taint seed. In our running example, cmd is a constraint variable used in recv, causing Saluki to identify the proper memory location corresponding to the cmd argument in all terms named recv. As is customary, Saluki uses unique identifiers to identify each taint seed.[9]

μflux: μflux is implemented as a custom interpreter that runs random parts of CFG and propagates the taint across the memory. μflux starts taint propagation from the instructions picked from specification and marked during the seeding. Than it executes parts of CFG randomly. μflux explores the paths regardless of the branch predicate and ignores context of data.

μflux stops execution when:

- Pre-defined max. instructions number was excited
- Saluki calls dynamically linked external function
- Saluki hits a jump with an indirect target

The Saluki Solver: In this step Saluki starts to follow the taint and tries to prove all properties specified in a policy. Saluki is constructive: it does not just show that there is

[6]BAP - BinaryAnalysisPlatform - https://github.com/BinaryAnalysisPlatform/

a violation, but gives the specific path and data dependencies used to show the property can be violated.

V. Results and tests

Saluki research group tested Saluki on many binaries in order to check its performance. No only did they find two binaries where srand seed was time dependent and 6 new zero-days bugs in 5 COTS products, but they even discovered Heartbleed in OpenSSL. Unfortunately in the same library Saluki showed 4 false positive alerts. In Lighthttp Saluki research team found command injection to system, buffer overflow due to wrong usage of strcpy. Apart from OpenSSL and Lighthttp they found 3 SQL injections in some COTOS products and many more.

This experiments shows that Saluki may be useful but the user needs to be aware that a false-positives may occur and that is why it is crucial to check every alert.

VI. Conclusions

Saluki is not sound nor complete. Its main advantage is that in comparison to other tools used to find vulnerabilities it is extremely fast. Unfortunately the authors of Saluki did not benchmark it against other tools which I think is a big drawback of the paper.

Sadly Saluki is not further developed and after publication the paper [9] there are no new commits in a Saluki's github repository.

References

[1] The MITRE Corporation. Common weakness enumeration.

[2] Dorothy Mckinney. Impact of commercial off-the-shelf (cots) software on the interface between systems and software engineering. pages 627–628, 01 1999.

[3] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[4] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. Anti-taint-analysis : Practical evasion techniques against information flow based malware defense. 2007.

[5] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3858 LNCS:124–145, 2006.

[6] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 797–812, Washington, DC, USA, 2015. IEEE Computer Society.

[7] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, and Haibing Guan. Static program analysis assisted dynamic taint tracking for software vulnerability discovery. *Computers & Mathematics with Applications*, 63:469–480, 2012.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, May 2008.

[9] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. Saluki: Finding taint-style vulnerabilities with static property checking. 01 2018.

[10] A. R. Bernat and B. P. Miller. Structured binary editing with a cfg transformation algebra. In *2012 19th Working Conference on Reverse Engineering*, pages 9–18, Oct 2012.

[11] P. P. F. Chan and C. Collberg. A method to evaluate cfg comparison algorithms. In *2014 14th International Conference on Quality Software*, pages 95–104, Oct 2014.

[12] J. Kim, T. Kim, and E. G. Im. Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*, pages 269–272, Sep. 2014.

[13] Muhammad Adnan, Faisal Aslam, Zubair Nawaz, and Syed Mansoor Sarwar. Rubus: A compiler for seamless and extensible parallelism. *PLOS ONE*, 12:e0188721, 12 2017.

[14] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.

[15] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. volume 6806, pages 463–469, 01 2011.

[16] David Brumley and Edward J Schwartz. BAP : A Binary Analysis Platform 3 Simple Lines. *Computer aided verification*, pages 463–469, 2011.