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. That is a big issue because analysis of binaries is far more complicated. To protect the privacy and security of the users and to analyse/check the binaries shipped by the vendors, some new tools and techniques for finding vulnerabilities in COTS need to be introduced. [3]

II. THEORETICAL INTRODUCTION

To understand steps of execution and Saluki's internal architecture the following introduction needs to be done.

A. Control Flow Graph - CFG

Representation, using graph notation, of all paths that might be traversed through a program during its execution including all the jumps, all the reads from the registers is called Control Flow Graph (Fig. 1). CFG shows all of the possible states of the program execution and the possible paths to achieve that state.

B. Approximation

Approximation stands for anything that is similar but not exactly equal to something else. In Fig. 2 a set of all possible states that can be achieved by the program during its execution is assigned with a letter A. To cover that set all the possible inputs needs to be supplied to the program, which will result in all of the possible states in the output. In other words, if all the possible inputs were supplied to the program, all the states from CFG will be reached. Unfortunately it is not possible to supply all of the potential inputs which is the reason for using the approximation.

[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]

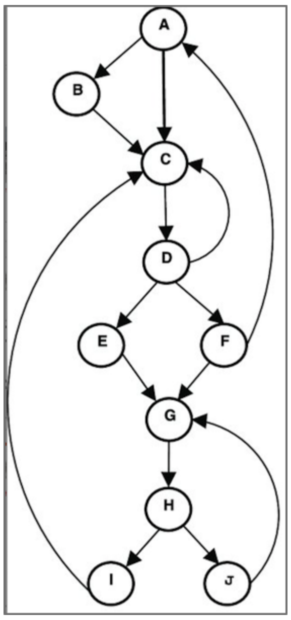

Fig. 1: Sample Control Flow Graph [4]

C. Under Approximation

Under approximation means that only the part of the possible states during the execution of the program are supplied. In that case it is not possible to cover all of the inputs but only the subset of them. The result of it is the subset of all the possible outputs. Under approximation in Fig. 2 is assigned with UA and is a subset of A. [5]

D. Over Approximation

Over approximation in case of achieving the CFG states, means that the set of the states in over approximation is bigger than the set of the all possible states that can be achieved by the program in CFB. Over approximation in Fig. 2 is assigned with OA and A is a subset of OA. [5]

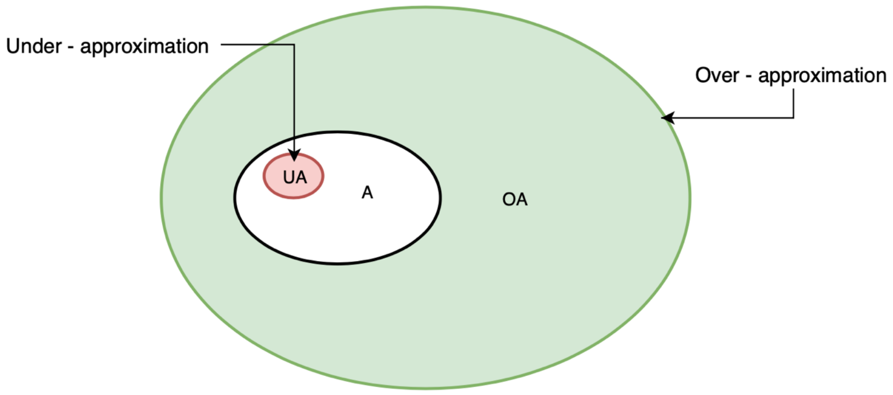

Fig. 2: Visualisation of over and under approximation [4]

E. Mixed Approximation

Mixed approximation is a subset of the states that are reachable by the program and the states that are not possible to be reached in CFG.

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. One goal of taint analysis is to identify data flows from attacker-controlled sources to security-sensitive sinks that do not undergo sanitization.[6] [7]

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. 3, 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 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.

A. Dynamic taint-style analysis

Dynamic taint-style analysis is the analysis done during the run time. Program is run and a few possible inputs are supplied, every time a binary needs to be rerun (always starts at the beginning of CFG). After that, tainted data triggered by the program at the run time, is followed. Dynamic taint-style analysis is an example of under approximation, because all of the states are reached during the run time of the program - are part of CFG. [7][8]

- 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

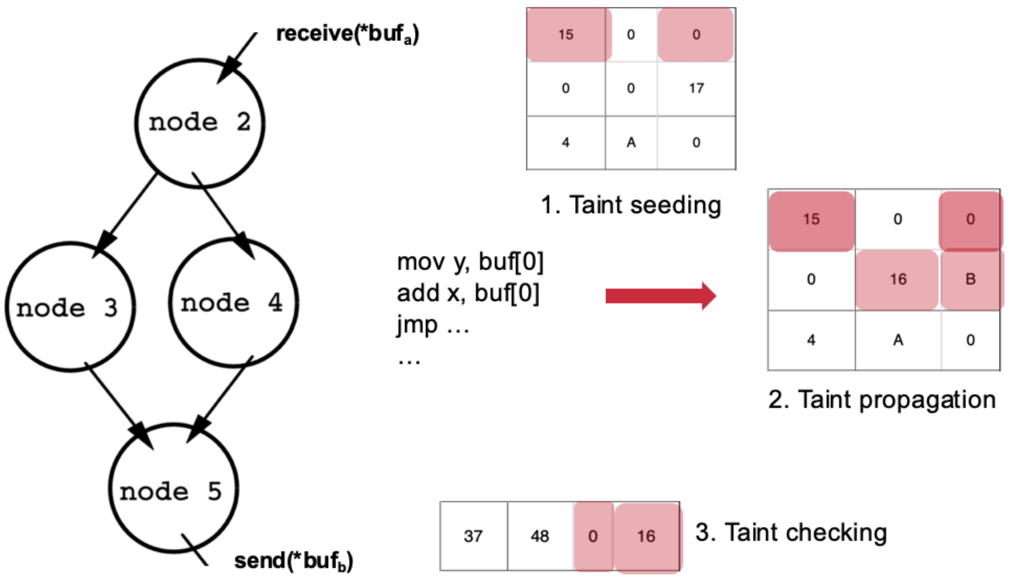

Fig. 3: Visualisation of taint analysis

B. *Static taint-style analysis*

Static taint-style analysis is an example of over approximation. Static analysis in contrary to the dynamic one is not done during the run time. 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

IV. Saluki Architecture

In contrary to a static and a dynamic taint-analysis 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.

Saluki does not need to start every time from the beginning. Additionally randomisation of an execution point, lets Saluki go deeper into the CFG than during a normal dynamic analysis. 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.

Saluki's execution can be divided into five steps. Its overall architecture is shown in a Fig. 4. [3] :

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.

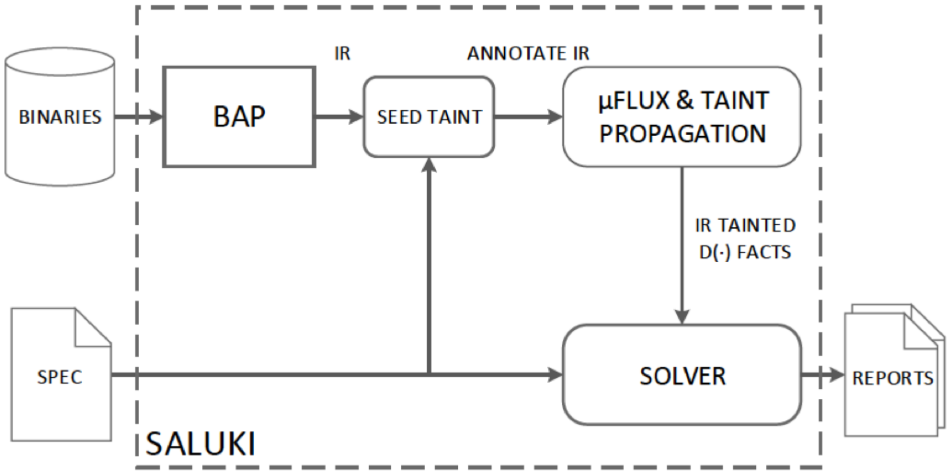

Fig. 4: Saluki Architecture [3]

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

[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

Binary processing: The binary is loaded to the Saluki. Than Saluki runs a BAP[6] (BinaryAnalysisPlatform) an open-source plugin-based binary analysis framework. BAP converts binary code into control flow graph - intermediate representation.[9]

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.[3]

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 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.

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

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

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

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

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

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