June 21, 2018

Flow-graph analysis of system calls for exploit detection

Anthony Desnos

Elena Petrova

Alexandre Boulgakov

Richard Neal

Zubin Mithra

Flow-graph analysis of system calls for exploit detection

ABSTRACT

One technique of improving computer security is to test an executable for presence of malicious code without running the executable. The present disclosure enables such detection of malicious code by leveraging the observation that system calls (syscalls) are a main pathway for exploits, since syscalls are an important way for a program to interact with an operating system kernel. The disclosure describes techniques to compute a control flow graph for the executable comprising only syscalls. A number of independent control flows are produced from such a control flow graph. Graph analysis/matching techniques are applied to detect exploit patterns in these syscall graphs, e.g., based on matching against known syscall exploit sequences for different vulnerabilities. In this manner, a potentially malicious executable is detected and can be isolated without exposing a computer system to damage.

KEYWORDS

- system call
- syscall
- static analysis
- exploit detection
- control flow graph
- kernel
- OS vulnerability
- malware detection

BACKGROUND

One technique of improving computer security is to test an executable for presence of malicious code without running the executable. Examples of malicious code include code that attempts to exploit known vulnerabilities within an operating system.

DESCRIPTION

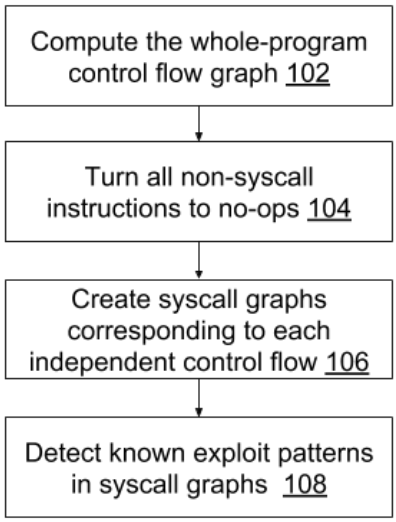

Fig. 1: Detecting known exploit patterns in binary executables

Fig. 1 illustrates detection of known exploit patterns in binary executables without running the executable. Static analysis is used to compute the whole-program control flow graph (CFG) (102) for the given binary file. Whole-program means the CFGs corresponding to called functions are embedded into the overall CFG such that entire system behavior is captured in a single graph.

All instructions other than those resulting in system calls (syscalls) are turned into no-operations (no-ops) (104). Syscall instructions include, e.g., architecture specific instructions

such as SYSCALL, SYSENTER, SVC, SWI, etc., and also include call or branch instructions to dynamically linked code, e.g., calls to known functions in dynamically linked libraries, identified by their names.

The result is a control flow graph comprising only syscalls. This graph encodes a superset of all syscall sequences that the given binary can result in. A number of graphs are produced from the binary, each corresponding to a potential independent control flow (106). While these are notionally separate graphs, since they are generated from the same executable, they may have significant overlap. Techniques of graph matching via merging, described in [1], can be applied, resulting in a single graph with multiple rooted subgraphs.

A syscall graph is produced for each entry point. Such entry points include, e.g., exported functions for a library or the entry point for an executable, and any functions that are passed to threading libraries to be run asynchronously. There is a one-one correspondence between graphs and entry points such that each graph is rooted at a single entry point.

Example: An executable includes the function `foo()` which calls the function `bar()`, and is defined as follows.

```
foo(){ bar();}
```

Both `foo()` and `bar()` are entry points. A graph corresponding to `foo()` is produced that is rooted at "`foo`" and contains both "`foo`" (as the root) and "`bar`". A graph corresponding to `bar()` is produced that is rooted at "`bar`".

Depending on the context the binary is used in when it is executed, syscall sequences from a control flow corresponding to a syscall graph may be interleaved, and multiple instances of each flow can be interleaved with each other as well.

Since syscalls are the main way for a program to interact with the OS kernel, many exploits rely on a particular sequence of syscalls to get the kernel into a particular bad state. Different sequences or patterns of sequences can be found, constructed or evaluated to determine whether they correspond to exploits for different vulnerabilities. Graph analysis and matching techniques are applied to detect known exploit patterns in the syscall graphs (108). In effect, specifications of a particular behavior of interest are matched against a superset of behaviors that are extracted from an executable file of interest using static analysis.

Different software implementations exploiting the same vulnerability perform the same actions, thus allowing detection of variations and different implementations of an exploit from the same patterns. Static analysis thus allows all potential behaviors of a binary to be explored efficiently. Analysis of syscall flow graphs allows specifically targeting program behaviors that are likely to be involved in kernel exploits and that must be present in a malicious binary file.

In this manner, the techniques of this disclosure can be applied to analyze executable or shared library files to determine if such executables include malicious code that attempts to exploit known vulnerabilities. Such data can be used in making security-related decisions about the executables. Online application stores or other platforms that offer third-party applications or software can use the techniques to test software that is submitted for download or sale to end-users. The techniques also find applicability in the computer security industry, e.g., within anti-virus, anti-malware, and other security products.

CONCLUSION

One technique of improving computer security is to test an executable for presence of malicious code without running the executable. The present disclosure enables such detection of malicious code by leveraging the observation that system calls (syscalls) are a main pathway for exploits, since syscalls are an important way for a program to interact with an operating system kernel. The disclosure describes techniques to compute a control flow graph for the executable comprising only syscalls. A number of independent control flows are produced from such a control flow graph. Graph analysis/matching techniques are applied to detect exploit patterns in these syscall graphs, e.g., based on matching against known syscall exploit sequences for different vulnerabilities. In this manner, a potentially malicious executable is detected and can be isolated without exposing a computer system to damage.

REFERENCES

[1] Boulgakov, Alexandre, "Efficient multi-graph or rooted subgraph matching via merging," Technical Disclosure commons, (June 18, 2018). Available online at https://www.tdcommons.org/dpubs_series/1252

[2] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3: A modern refinement checker for CSP," *Tools and Algorithms for the Construction and Analysis of Systems* pp. 187-201. Available online at http://www.cs.ox.ac.uk/projects/fdr/

[3] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, A. W. Roscoe, "Modelling and analysis of security protocols: the CSP approach," Addison-Wesley (2001)

[4] A. Boulgakov, T. Gibson-Robinson, A. W. Roscoe, "Computing maximal weak and other bisimulations", *Formal Aspects of Computing 28(2)* pp. 381-407

[5] A. Boulgakov, "Improving scalability of exploratory model checking," PhD thesis, University of Oxford, 2016. Available online at https://ora.ox.ac.uk/objects/uuid:76acb8bf-52e7-4078-ab4f-65f3ea07ba3d