December 2022

Dynamic Real-time Verification of Program Call Flows

Nic Watson

Chris Schneider

Dynamic Real-time Verification of Program Call Flows

ABSTRACT

This disclosure describes a dynamic overlay that delineates control flows of an application to identify unexpected code execution pathways that may be indicative of a security breach. Identifying such unexpected (or unauthorized) execution pathways can enable their prevention. The overlay is generated by observing the control flow through the application to produce a histogram of probabilities from a first function call to subsequent function calls. Using the overlay enables the detection of attacks with higher fidelity and at a lower cost than existing approaches.

KEYWORDS

- Security breach
- Control flow graph
- Call stack
- Control flow integrity
- Stack inspection
- Execution pathway
- Return-oriented programming (ROP)
- Blind return-oriented programming (BROP)

BACKGROUND

Execution of unexpected pathways within a given piece of computer code is a tell-tale sign of a security breach. For example, return-oriented programming (ROP and its variant, blind ROP) is an exploit where an attacker gains control of the call stack to alter the control flow of a program and execute instruction sequences already present in the memory, allowing the attacker to perform arbitrary operations on the compromised machine.

DESCRIPTION

This disclosure describes a dynamic overlay that delineates control flows of an application to identify unexpected code execution pathways. Identifying such unexpected (or unauthorized) execution pathways can ultimately result in their prevention.

The overlay is generated by observing the control flow through the application to produce a histogram of probabilities from a function call A_1 to any subsequent call A_z. Using the overlay enables detection of attacks with higher fidelity and at a lower cost than existing approaches.

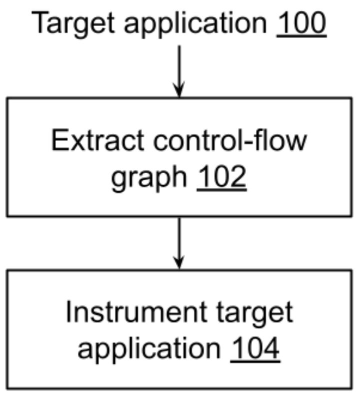
Target application 100
Extract control-flow graph 102
Instrument target application 104

Fig. 1: Preparing a target application for control flow integrity

As illustrated in Fig. 1, a target application (100) is prepared for real-time verification of API call flows, e.g., for control flow integrity, as follows. A control flow graph (102) is extracted from the target application. Such extraction is required once per version of the application, e.g., not at every run of the application. The target application is instrumented (104) such that its function calls during runtime can be sent to an observer process running in parallel to the target application.

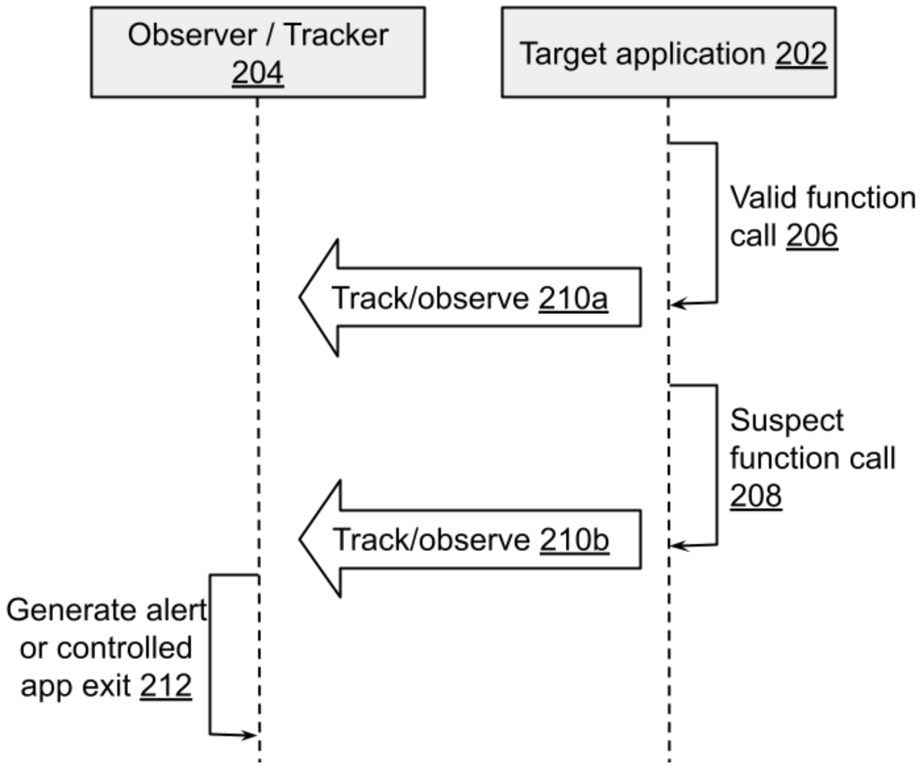

Fig. 2: Dynamic, real-time verification of API call flows

Fig. 2 illustrates dynamic, real-time verification of API call flows of a target application (202), instrumented as described above and observed by a tracker process (204). As the target application runs, it generates a number of function calls, some valid and some suspect, e.g., indicative of an attack or security breach. The instrumentation on the target application enables the observer/tracker process to track or observe the function calls (210a-b) which may occur in any order.

A valid function call (206), e.g., a call specified in the code by the developer, is observed without reaction, but a suspect function call (208) causes the observer/tracker process to generate an alert or a controlled app exit (212). Some examples of suspect (or otherwise interesting) function calls include:

- Function calls whose input and output destinations have a low likelihood. The generation of function call likelihoods is described below.
- Function calls that access sensitive application programming interfaces (APIs), user data, or other artifacts.
- Function calls not specified in the code by the developer. If, for example, in the code, function-A is called exclusively by function-B, a runtime call to function-A that does not originate from function-B (or its encapsulating class or module) is suspect or at least of interest.
- Function calls that occur from unexpected locations in memory (call frames), based on the first call to the function upon program instantiation.
- Return values of any stack frame that differ from prior function calls originating from the same call site.
- Any other static (or discrete) rule as specified by the code developer or tester.

Observing and tracking of suspect or interesting function calls occurs in a process (the observer/tracker) distinct from and parallel to the target application, e.g., read and write duties are separated between the observer and the target. Such separation ensures that an attack is actually prevented; integration of the observer functionality into the target application might slow down the application but not prevent an attack. The observer/tracker uses the extracted control flow graph (as illustrated in Fig. 1) to determine if runtime function calls executed by the target application conform to the graph. Violations of the graph generate alerts or controlled application exits.

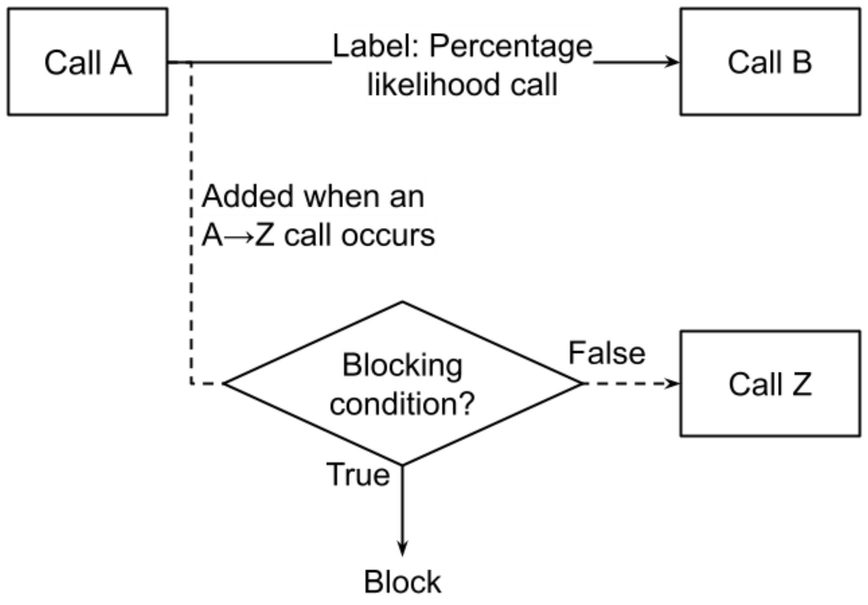
Call A
Label: Percentage likelihood call
Call B
Added when an A→Z call occurs
Blocking condition?
False
Call Z
True
Block

Fig. 3: Development of call likelihood probabilities and testing for blocking conditions

Fig. 3 illustrates the development of call likelihood probabilities and testing for blocking conditions. A histogram of calls between functions is maintained. The event of a call from a function A to a function B results in an update within the histogram of the relative frequency (likelihood) of the A→B call. Similarly, a call from A to Z results in the addition or update of the relative frequency of the A→Z call. Before the histogram is updated, the function call is tested for blocking conditions, e.g., a function call with a signature of an exploit, as described above. If a blocking condition is met, then the call is blocked, an alert is issued, or a controlled exit is executed.

In this manner, the described techniques can rapidly detect control flow violations that originate from memory corruption, buffer overflow, or other types of exploits by:

1. extracting a control flow graph for a given program
2. labeling its edges with transition probabilities
3. identifying calls that deviate from the labeled control flow graph, and

4. upon detection of deviated calls, blocking the calls or by issuing an alert.

In an implementation with controlled exits, e.g., shadow stacks, the techniques detect and guard against ROP or BROP activity. The techniques provide an effective mechanism for deep inspection of the call stack and for maintaining the integrity of the control flow.

CONCLUSION

This disclosure describes a dynamic overlay that delineates control flows of an application to identify unexpected code execution pathways that may be indicative of a security breach. Identifying such unexpected (or unauthorized) execution pathways can enable their prevention. The overlay is generated by observing the control flow through the application to produce a histogram of probabilities from a first function call to subsequent function calls. Using the overlay enables the detection of attacks with higher fidelity and at a lower cost than existing approaches.

REFERENCES

[1] Ben Niu and Gang Tan, "Per-input control-flow integrity," available online at https://www.cse.psu.edu/~gxt29/papers/picfi.pdf accessed Nov. 30, 2022.

[2] "Blind return-oriented programming," available online at https://en.wikipedia.org/wiki/Blind_return_oriented_programming accessed Nov. 30, 2022.