Subject: [NDSS 2018] Early rejection notification for paper 160
From: "NDSS 2018 HotCRP" <noreply@ndss18.hotcrp.com>
Date: 09/19/2017 01:00 AM +0000
To: Paul Muntean <paul@sec.in.tum.de>
CC: alina.oprea@gmail.com, traynor@ufl.edu

Dear Paul,

We regret to inform you that your paper 160 has not been selected to be
included in the 2018 Network and Distributed System Security (NDSS)
Symposium.

Your reviews are included below. We wish you the best of luck with this
work in the future, and hope that the comments here are as helpful as
possible to revise your paper. Thank you for submitting to NDSS 2018!

Regards,
Patrick Traynor and Alina Oprea
NDSS 2018 Program Co-Chairs

Review #160A
===

Overall merit
----
2. Weak reject

Relative Ranking
----
3. Top 25% but not top 10% of reviewed papers

Reviewer expertise
----
4. Expert

Writing quality
----
4. Well-written

Paper summary

This paper presents a binary analysis that recovers function types for functions
and indirect function calls. The binary is then instrumented with a CFI policy
that enforces checks for the forward edge. In addition to the forward edge, the
backward edge is protected through a similar CFI, target-based matching
mechanism.

Comments for author

+ Binary analysis to infer function arguments at call sites

+ Low overhead

- Easily bypassed defense on the backward edge

- Same policy as "Strict Virtual Call Integrity Checking for C++ Binaries" by Elsabagh et al, AsiaCCS'17 for the forward edge.

The proposed analysis is very similar to the binary analysis proposed in [18] (VCI -- Virtual Call Integrity). The main difference to 18 is a policy for return instructions that is unfortunately weak and can be mitigated.

The backward edge policy supports three modes: range check, label, and precise lookup. It is interesting to see a discussion about the trade-offs. A range-check, is low overhead but provides the attacker many different target locations to branch to while a label may be imprecise due to collision as each function can only have a single label. The most precise check results in several lookups and therefore leads to more overhead. Unfortunately, all backward-edge checks are imprecise and overly permissive.

For the backward edge policy: Control-Flow Bending by Carlini et al has shown that a CFI-based policy on the backward edge is not precise enough to stop attacks. That's why CFI defenses focus on the forward edge, leaving the backward edge for alternate defenses such as shadow stacks.

LLVM-CFI enforces a similar policy (i.e., same class-based precision) as vTrust at similarly negligible overhead and is deployed in practice in the LLVM compiler and available by toggling a compiler switch. The implementation is mature and has been evaluated widely.

"It is important to note that there are no C++ language semantics which can be used to enforce type and parameter count matching for indirect caller/callee pairs, this could be addressed with specifically intended language constructs in the future"

I strongly disagree. The C++-based solutions leverage the precise number of parameters and their types when building the allowed target set. These properties are enforced through LLVM-CFI, for example.

"Therefore, we can state that our structural matching between
ground truth and TYPE SHIELD s callsites is almost perfect"
Almost leads to imprecision and may result in false positives or false negatives. What leads to the differences?
Also, I assume that the analysis only evaluates indirect targets. I'd like to see a breakdown of the errors and matched targets based on number of functions.

The performance evaluation is odd, what are the relative numbers on the graph? (Fig6)

#1: such ***as*** dynamic dispatch

Questions for authors' response

Please address the disparity between the claim about C++ semantics and not having any defenses leveraging C++ types/number of arguments for indirect calls (as, e.g., LLVM-CFI does in its open source implementation).

Explain the differences for the forward edge policy compared to [18].

Please elaborate on the differences in Table I; why are the numbers different
from the ground truth.

* *

Review #160B
==========

Overall merit
---
2. Weak reject

Relative Ranking
---
2. Top 50% but not top 25% of reviewed papers

Reviewer expertise
---
4. Expert

Writing quality
---
4. Well-written

Paper summary

This paper proposes a software defense for hardening binaries against code-reuse attacks, such as Counterfeit Object-Oriented Programming (COOP). The presented approach, dubbed TypeShield, statically analyzes the program binary before enforcing a lightweight control-flow policy on C++ programs at run time. In particular, the policy includes a forward-edge and (three variants of) backward-edge protection, which limit the set of possible jump targets during program execution. The paper presents a prototypical implementation of the approach, which is evaluated against several server applications and the SPEC2006 benchmarking suite, for which the geomean overhead of TypeShield lies around 4%.

Comments for author

Strengths:
- well written and easy to follow
- detailed evaluation

Weaknesses:
- missing conceptual novelty
- comparison with related work only shows implementation-specific differences
- does not mention the known shortcomings of binary defenses ([A,B])
- the remaining attack surface is never discussed (cf., Section VI.D in [20])

Questions for authors’ response

Strengths:
- well written and easy to follow

- detailed evaluation

Weaknesses:
- missing conceptual novelty
- comparison with related work only shows implementation-specific differences
- does not mention the known shortcomings of binary defenses ([A,B])
- the remaining attack surface is never discussed (cf., Section VI.D in [20])

Comments:
Code-reuse attacks such as return-oriented programming (ROP) and counterfeit object-oriented programming (COOP) have provided a broad field of research over the last decade and naturally many results have been published over time.
Code-reuse attacks generally require a memory-corruption vulnerability in the target program, which can be exploited by an adversary to hijack the control flow of the execution at run time.
This is usually done by overwriting a code pointer in program memory, and many approaches have been proposed to defend against such attacks.
Of these defenses, control-flow integrity (CFI) proved to offer very promising guarantees: CFI labels branch targets and enforces the correct control flow by checking the intended branch label and return address dynamically during execution.
However, CFI faces a number of challenges, such as the induced run-time overhead and the precision of the underlying label set.

The paper is generally well written and although the implementation section is very limited, offering practically no framework details, I like the wide range of presented evaluation results. As the main motivating factor of this paper is the COOP exploit, which was presented for browsers, I would have expected an evaluation of TypeShield, e.g., for Firefox. However, my main concern with this paper is that it is too similar to existing approaches, and provides no conceptual novelty compared to the current state-of-the-art. Additionally, the paper does not show any tangible improvement over the previously achieved results with related binary CFI schemes. In general, to achieve the necessary label precision, fine-grained CFI approaches use source code to accurately analyze a program's control-flow graph, and while many binary-CFI schemes have been proposed, these coarse-grained approaches were in fact shown to be vulnerable to a number of attacks [A,B].

For C++ programs, constructing an accurate control-flow graph from the binary representation is difficult, because of the dynamic dispatch semantics that are required to implement the object oriented programming paradigm and support, e.g., polymorphic operations. For this reason, recent approaches leverage knowledge about the (highly standardized) C++ run time environment and detailed static analysis of the binary representation. To this end MARX [19] demonstrated how to reverse-engineer information about the original program class hierarchy from a C++ binary. Given the dedicated comparison section in the beginning, I expected to see either some conceptual novelties or a performance improvement over the state-of-the-art, but in fact TypeArmor [20] already presented the same basic approach, i.e., matching information about the target function (such as the argument count) with the program's call-site information. While I realize that TypeShield offers a built-in backward edge p
 rotection, TypeArmor leverages existing defenses for this (i.e., [25]). Additionally, the overhead numbers for TypeShield are slightly higher than the related work, and hence, the overall contribution is too low for this venue.

Nits:

- everybody loves statistics, but the problem of memory corruption does not need a big

motivation, and hence, Figure 1 is really unnecessary

- I am missing a discussion about the drawbacks of binary CFI (e.g., [A,B])

- "can be used by analyst"

[A] Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, Davi et al., USENIX Security 2014

[B] Out of Control: Overcoming Control-Flow Integrity, Göktas et al., IEEE S&P 2014