# Lessons Learned

Anders Choi

September 2024

## 1 Industry Case Study

### 1.1 Offense: How would you attack Zoom's Linux client?

1. Code injection attack (via buffer overflow + ROP): Since Zoom's Linux binary file lacks DEP, ASLR, and Canaries, code can be injected directly into the stack and executed. Since the lack of DEP allows for the injection of custom code by the attacker, they could easily run commands on the host system running the Zoom client (at least as much permission given to the client), such as extracting sensitive information from the host system. Sensitive information could also trivially be extracted from the client. Client behavior can also be manipulated, such as granting unauthorized privileges to specific users or disrupting meetings.

2. Code reuse attack (via buffer overflow + ROP): With lack of DEP, the attacker could simply inject custom code as explained above, but code-reuse attacks are also possible (although less powerful than code injection attacks, as specific gadgets need to be found in order to execute exploitative commands). Specialized types of code-reuse attacks such as AOCR and PIROP is also possible, but would not be much help beyond a simple code-reuse attack, because there are no defense mechanisms such as ASLR in place to thwart it. Implications of the above attack (code-injection) also apply to this attack, as long as gadgets can be found to replicate them.

3. Format string attack: Without fortification, vulnerable format functions such as printf can be exploited by attackers. Memory read/writes, code execution, and DoS are possible intents of this attack.

### 1.2 Defense: How would you address the security issues pointed out by Mudge?

1. Migrate to more recent toolchains: While Mudge proposed a lot of recommendations such as writing security-related unit-tests and avoiding vulnerable functions, it seems that the biggest problem with the Linux Zoom

binary that Mudge was focusing on was its outdated toolchain, as the Windows and Mac versions did not have the same vulnerabilities presented in the Twitter thread.

2. Use analysis tools to identify vulnerabilities: Adopt static analysis tools such as Fortify into the pipeline to find vulnerabilities in the source code. Use dynamic analysis tools such as Valgrind to detect unusual patterns during runtime in testing and live environments.

3. Apply diversification techniques: Software diversification, while it does not completely eliminate the attack vectors, make it significantly more difficult to exploit existing vulnerabilities. For instance, even if basic security measures such as DEP and Canaries are not present, diversifying the software would make it difficult to pinpoint specific memory locations necessary for ROP attacks.

# 2 Academic Case Study

## 2.1 What extent does the presented approach, CROW, protect against the outlined threats effectively?

The paper mentions two types of attacks that threaten the security of WebAssembly softwares: ROP attacks and side-channel attacks.

CROW thwarts ROP attacks by not only diversifying the static code, but also ensuring that the stack traces of the diversified programs are significantly different from each other, making it harder to pinpoint specific locations in the stack necessary for ROP attacks. Differing static codes also means that the available gadgets vary widely across different program variants, making it difficult for attackers to retain information across variants.

Diversification in execution traces, according to the paper, is highly effective against side-channel attacks. This is because the timing of specific operations are seemingly randomized, making it harder for attackers to approximate the timing of certain operations they are trying to analyze. However, I suspect that the lack of higher-level diversification (function-level and program-level) would make CROW not as effective against side-channel attacks as it could have been.

## 2.2 How effective is CROW at diversifying WebAssembly programs? Think about the degrees-of-freedom!

According to the definitions in the paper, CROW generates variants for "blocks", which are groupings of instructions within a function. This means that the diversification happens only at the instruction level and the block level. The method could be improved by introducing diversification in higher levels such as function reordering.

### 2.3 Did you find any problems with the way CROW diversifies programs?

The fine-grained nature of the variants (not beyond the block level) limits the defense against side-channel attacks that CROW claims to be effective against.

### 2.4 Does the evidence provided by the paper give you confidence in the described approach?

Aside from the potential for applying additional types of diversification (in function level and program level), I was interested to see that the size of the diversified binary files stayed more or less the same as the original, with some variants becoming smaller. Although the paper didn't directly talk about the execution times of the variants, I suspect that the more-or-less constant size of the binary implies that the execution time overhead is not very high. I was fascinated by the novelty of using a superoptimizer for the purpose of diversification.