RUHR UNIVERSITÄT BOCHUM

RUB

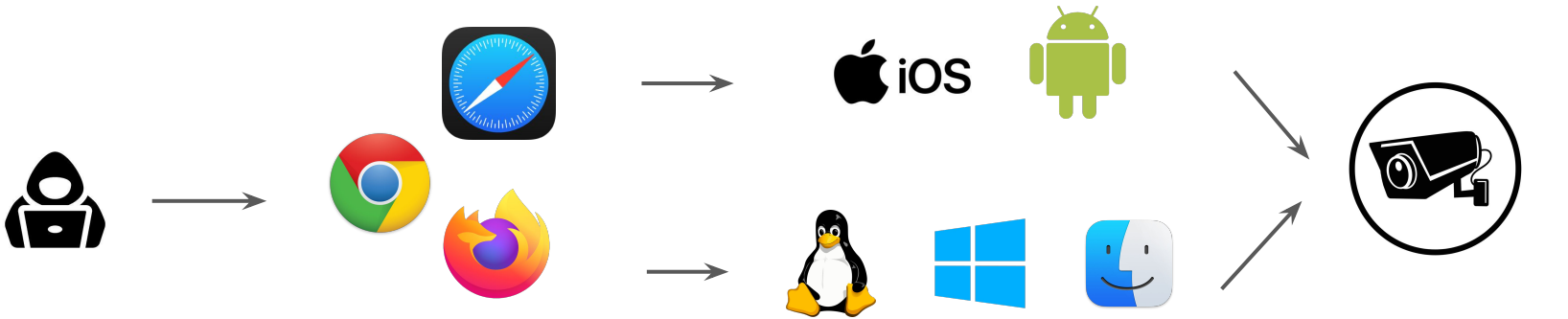
Squeezing Juicy *Variant Bugs* Out of Modern Browsers

Han Zheng, Flavio Toffalini, Qiang Liu, and Mathias Payer

EPFL

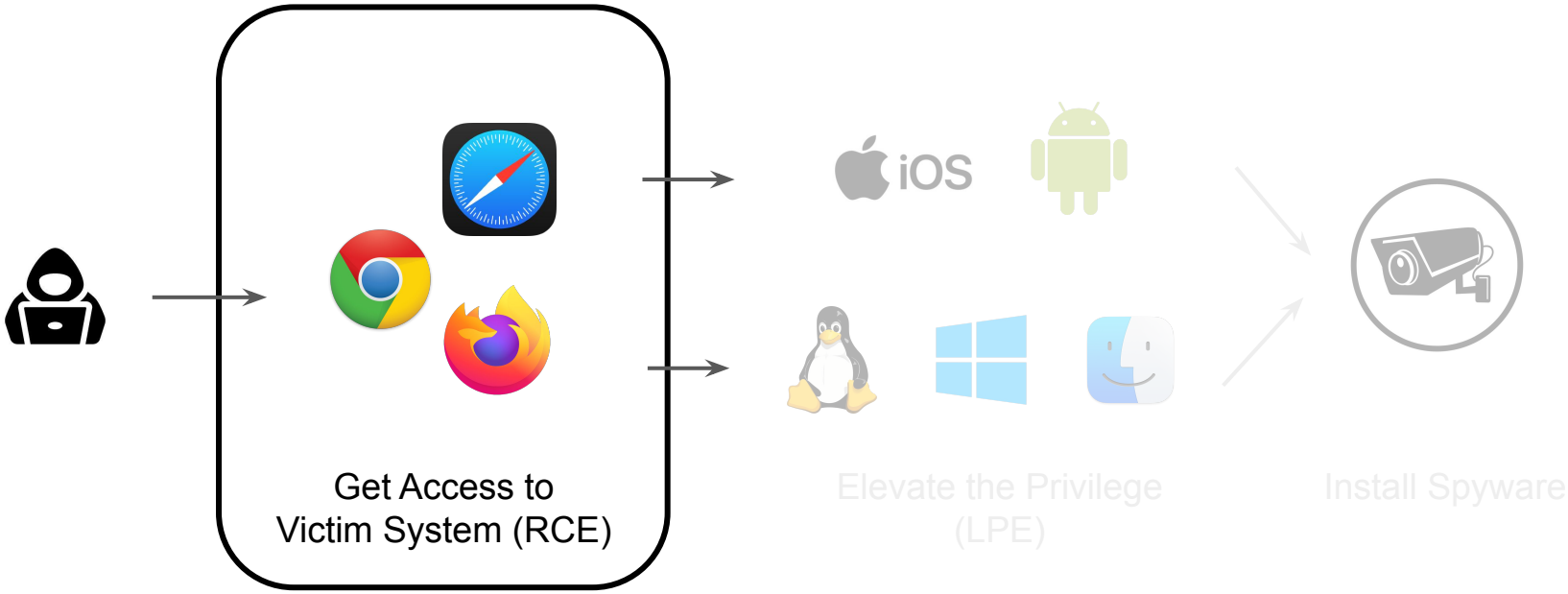
hexhive

Motivation: Browsers are Attack Entry Points



[1] Buying Spying: Insights into Commercial Surveillance Vendors, Google TAG

Motivation: Browsers are Attack Entry Points



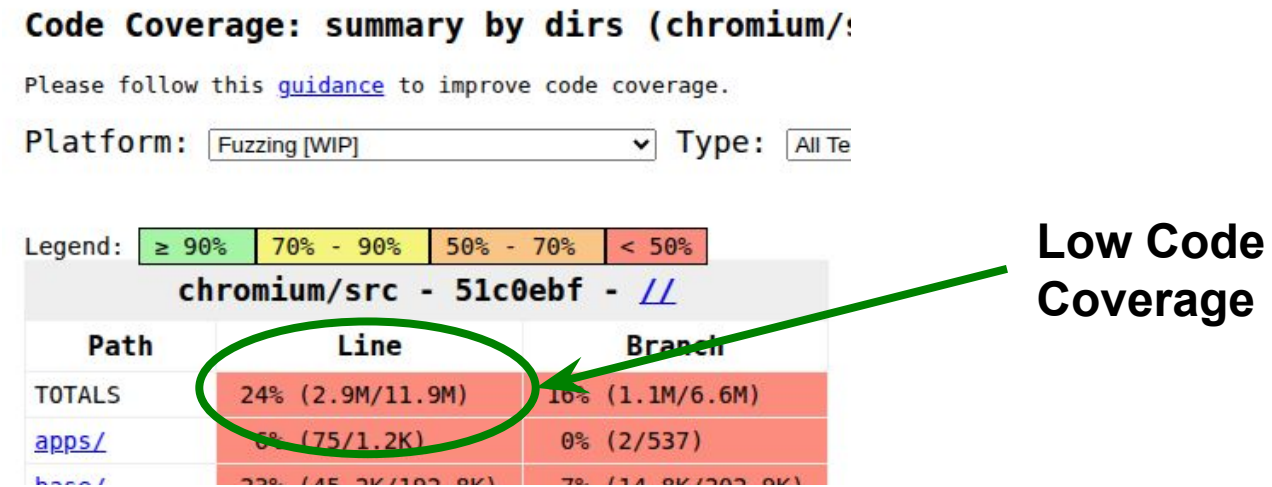
[1] Buying Spying: Insights into Commercial Surveillance Vendors, Google TAG

Challenges for Browser Vulnerability Finding

Challenge 1: Large Codebase Size (Where to look for bugs?)

Challenge 2: Cross-Context Interactions (How to model bugs?)

Challenge 3: Cross-Domain Dependency (Assess exploitability?)



[1] Chromium Fuzzing Coverage, https://analysis.chromium.org/coverage/p/chromium.

Challenges for Browser Vulnerability Finding

Challenge 1: Large Codebase Size (Where to look for bugs?)

Challenge 2: Cross-Context Interactions (How to model bugs?)

Challenge 3: Cross-Domain Dependency (Assess exploitability?)

Chromium C++ object lifetimes are often too complicated for human brains

– Top security things for Chromier to remember[1]

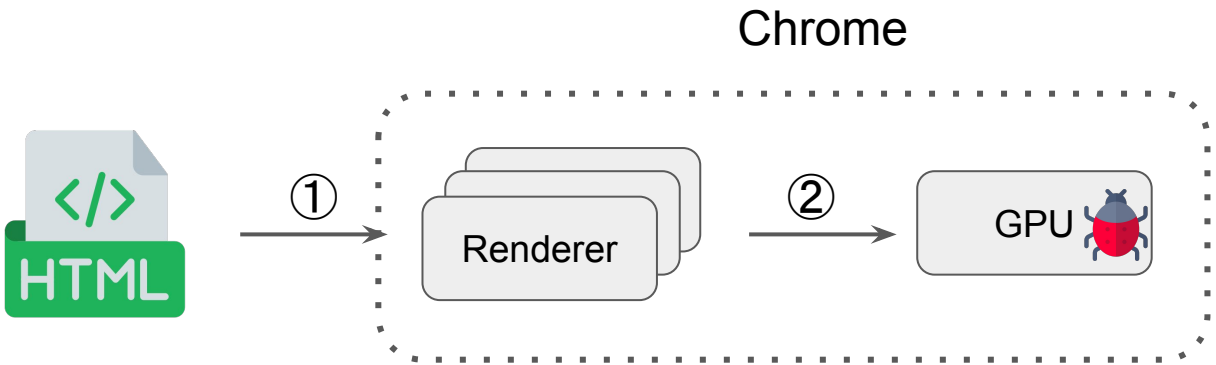
[1] Top security things for Chromies to remember, Chromium doc.

Challenges for Browser Vulnerability Finding

Challenge 1: Large Codebase Size (Where to look for bugs?)

Challenge 2: Cross-Context Interactions (How to model bugs?)

Challenge 3: Cross-Domain Dependency (Assess exploitability?)



① Compromise the renderer (achieving renderer RCE)
② Renderer sends crafted command via IPC and trigger overflow

[1] CVE-2024-8193, High-Severity Sandbox Escape in Chrome

Variant Analysis: The Up and Downs

Variant analysis ***hunts unknown bugs by abstracting known bugs***

Current practices do not scale:

- [High False Negatives] Ctrl+F “buggy code” only finds exact clones
- [Low Scalability] Manual auditing only finds one class of bugs[1]

Key question: **How can we find bugs *with low FN rate* and *without deep code understanding*?**

We propose Grape, *a formalized variant analysis workflow* that works for general variant bugs, with higher scalability and lower FN rate.

[1] Put in one bug and pop out more: An effective way of bug hunting in Chrome, BlackHat US 2021

Why not using grep to search the code snippet?

```
auto tokens = CSSTokenizer(selector_string).TokenizeToEOF();
// at this point, CSSTokenizer (temporary obj) is destroyed
// the tokens point to a freed object
 CSSParserTokenRange range(tokens);
// range is used, introducing UAF
```

Fix

```
// Fixed code
CSSTokenizer tokenizer(selector_string)
auto tokens = tokenizer.TokenizeToEOF();

CSSParserTokenRange range(tokens);
```

CVE-2024-7000 and its fix

Why not using grep to search the code snippet?

```
auto tokens = CSSTokenizer(selector_string).TokenizeToEOF();
// at this point, CSSTokenizer (temporary obj) is destroyed
// the tokens point to a freed object
 CSSParserTokenRange range(tokens);
// range is used, introducing UAF
```

Fix

```
// Fixed code
CSSTokenizer tokenizer(selector_string)
auto tokens = tokenizer.TokenizeToEOF();

CSSParserTokenRange range(tokens);
```

CVE-2024-7000 and its fix

```
std::optional<CSSParserToken> GetAttrSubstitutionValue(
    const String& attribute_value,
    const CSSAttrType& attribute_type,
    const CSSParserContext& context) {
```

```
 CSSTokenizer tokenizer(attribute_value);
 auto tokens = tokenizer.TokenizeToEOF();

 return tokens;
}
```

Another variant, is this safe?

Why not using grep to search the code snippet?

```
auto tokens = CSSTokenizer(selector_string).TokenizeToEOF();
// at this point, CSSTokenizer (temporary obj) is destroyed
// the tokens point to a freed object
 CSSParserTokenRange range(tokens);
// range is used, introducing UAF
```

Fix

```
// Fixed code
CSSTokenizer tokenizer(selector_string)
auto tokens = tokenizer.TokenizeToEOF();
```

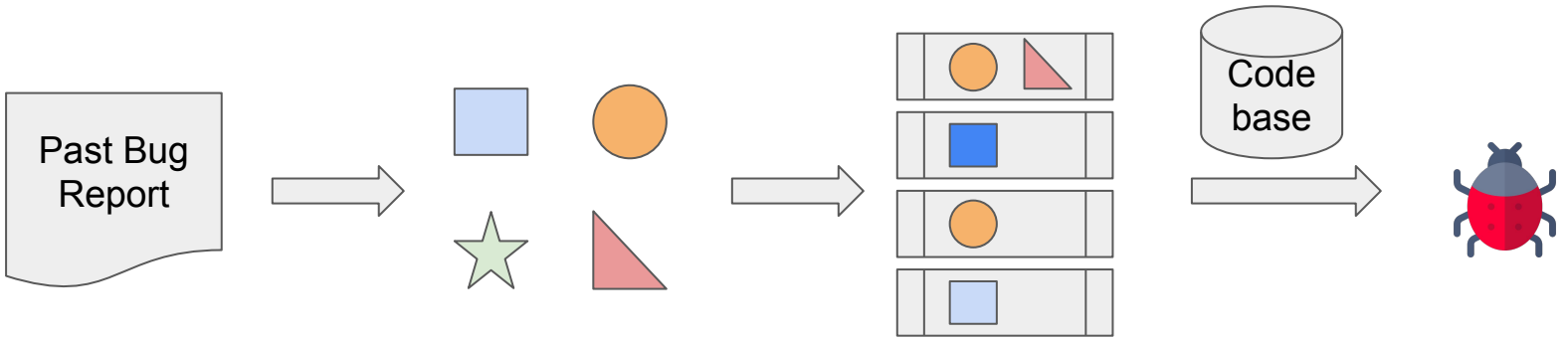
```
std::optional<CSSParserToken> GetAttrSubstitutionValue(
    const String& attribute_value,
    const CSSAttrType& attribute_type,
    const CSSParserContext& context) {

 CSSTokenizer tokenizer(attribute_value);
 auto tokens = tokenizer.TokenizeToEOF();

 return tokens;
}
// use of returned tokens is abuse
```

Some fix-alike code is not safe!
Without bug model comprehension, it is hard to distinguish them

Grape: Overview

Grape: a variant analysis model to simplify audits

1) **classify** variant bugs into common types
2) **model** variant bugs using four key elements
3) **formalize** a workflow to find new variant bugs



Grape: Variant Bug Model

Our variant bug model includes four elements:

- ***Context:*** the function or class method that contains the bug
- ***Assumption:*** the program's *expected behavior* in the developer's mind
- ***Violation:*** the code that violates the developer's expectation
- ***Abuse:*** the vulnerability triggered following the violation of the developer's expectation.

```
// Context: CFFL_ListBox::SaveData()
void CFFL_ListBox::SaveData(...) {
   ObservedPtr observed_box(pListBox);
   // Violation: Free pListBox
   m_pWidget->SetOptionSelection(i);
   // Assumption: pListBox not freed
+    if (!observed_box) {
+        return;
+    }
   // Abuse: Use-After-Free
   pListBox->IsItemSelected(i);
}
```

CVE-2023-2932, an UAF in PDFium

Evaluation: New Bug Findings

- We leverage Grape to find new bugs in security-critical software like Chromium, VSCode, and OpenSSL.
- Our findings covers temporal and spatial memory corruptions (use-after-free, use-after-return, heap buffer overflows), and logical flaws (Information Leaks), totaling 24 new bugs.
- Six bugs were awarded bug bounties from the Chrome Vulnerability Reward Program

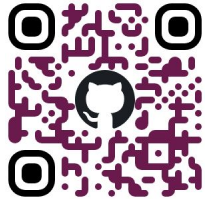
Conclusion & QA

Variant analysis sits between a dynamic and static approach

We formalize the variant bugs with four key elements (context, assumption, violation, abuse)

We conclude how to detect four variant bug types (code clone, similar caller, same callee, same callee chain) with our model

We found 24 bugs in security-critical systems and received bounties



EPFL RUHR UNIVERSITÄT BOCHUM RUB PIZZA LAB hexhive