École Polytechnique Fédérale de Lausanne

Exploiting Chromium with the help of type++

by Damiano Amatruda

Master Research Project

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Nicolas Badoux
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

9 July 2023

Abstract

Type confusion is a critical category of bugs commonly found in C++ programs which enables attackers to manipulate objects by misinterpreting their types. Type++ is a novel C++ dialect which prevents type confusions by incorporating Run Time Type Information (RTTI) into every object involved in a cast operation. In this project we aim to assess the compatibility of type++ with the Chromium browser, a widely-used and complex software application. Our focus primarily revolves around analyzing the most reported type confusions within this context.

Contents

Chapter 1

Introduction

Type confusion bugs are misinterpretations of object types that can have severe consequences [11], as they can enable attackers to compromise user data and execute harmful code through various forms of attack, including memory corruption and arbitrary code execution [10] [17] [22] [7].

To address these vulnerabilities, a new C++ dialect called type++ [1] has been developed. Type++ takes a proactive approach by incorporating Run Time Type Information (RTTI) into every object involved in a cast operation. This enables the dialect to validate the correctness of each cast and possibly also terminate the program immediately upon detecting a type confusion, preventing further exploitation.

The objective of this project is to assess the compatibility and impact of type++ in the context of the Chromium browser, a widely-used and complex software application. Specifically, we analyze the most reported type confusions in its codebase. During our analysis we discovered an edge case that was not handled properly in type++ and fixed it.

Chapter 2

Background

2.1 Type Confusion and The C++ Programming Language

Type confusion is a critical class of vulnerabilities that can arise in programming languages when an object is mistakenly interpreted as belonging to an incompatible type [11]. C++ is an object-oriented programming language particularly susceptible to type confusion due to its strong ties to C. It introduces additional features such as classes, which represent types, and supports inheritance and polymorphism, which make it a powerful language for building complex software systems.

In C++, inheritance enables the creation of parent-child relationships that allow derived classes to inherit properties and behaviors from their base classes. On the other hand, polymorphism allows objects of different classes to be treated uniformly through a common interface. This is achieved through virtual functions, defined in base classes and overridden in derived classes, and dynamic dispatch, where the appropriate function implementation is determined at runtime based on the actual type of the object.

A crucial component in C++ that enables polymorphism and dynamic dispatch is the vtable pointer [9]. Vtable pointers help identify the type of an object and allow for proper function resolution at runtime. They are pointers stored within objects and point to the corresponding vtable (or virtual function table), which contains Run Time Type Information (RTTI) and function pointers for the object's class. By examining the vtable, C++ can determine the object's type and invoke the appropriate virtual function.

However, not all C++ objects have vtable pointers. C++ classes are interoperable with C structures. Non-polymorphic classes, including C-style structures, do not have vtable pointers. As a result, runtime type checking is only possible for polymorphic classes, which limits the scope of type checking in C++.

2.1.1 C++ Cast Operations

C++ provides various cast operations that allow developers to convert between types:

- `const_cast` is unrelated to type confusion and is considered out of scope for this project.
- `dynamic_cast` enables type conversion with runtime verification. It relies on the presence of RTTI and can only be used for polymorphic types.
- `static_cast` performs type conversions without runtime checks. It does not require RTTI and provides limited compile-time validation. If used incorrectly, `static_cast` can introduce type confusion vulnerabilities, particularly when casting pointers from outside the class hierarchy.
- `reinterpret_cast` disregards compile-time and runtime verification, reinterpreting the memory area of an object as the target type. It delegates the verification completely to the programmer. If the types are not aligned in memory, it leads to security issues.

In contrast to C, where casts primarily determine field offsets, C-style casts in C++ are replaced by the cast operators `const_cast`, `static_cast` and `reinterpret_cast` in this order of priority. The first cast operator that satisfies compilation requirements is selected.

2.1.2 C++ Type Confusions and Exploitations

Type confusions have been identified as the Common Weakness Enumeration *CWE-704: Incorrect Type Conversion or Cast.* They are categorized in two ways: *Down-to cast* and *Unrelated cast* (or `void*` cast).

Down-To Cast Type Confusion

A down-to cast type confusion occurs when an object is cast to an incompatible derived type. It is only safe when the object being cast is an instance of the derived class or one of its subclasses. If the object being cast is not of the expected type, it can allow attackers to execute arbitrary commands.

Exploitation. Listing 1 shows an example exploit. In this scenario, an object of base class `Base` is cast to a child class `ChildPrinter` and the method `print` is called on it. The exploit creates an instance of `ChildExecutor`, a different child class of `Base`, and passes it to the vulnerable function, which mistakenly treats it as an instance of `ChildPrinter` and thus calls the method `exec` instead of `print`.

```cpp
#include <iostream>
class Base {};
class ChildPrinter : public Base {
public:
    virtual void print(const char *str) { std::cout << str << std::endl; }
};
class ChildExecutor : public Base {
public:
    virtual void exec(const char *cmd) { system(cmd); }
};
void downToCast(Base* b, const char *msg) {
    static_cast<ChildPrinter*>(b)->print(msg);
}
void exploit() {
    Base *b = new ChildExecutor();
    downToCast(b, "/bin/sh");
    delete b;
}
```

Listing 1: Example of an exploit of a down-to cast type confusion

Unrelated Cast Type Confusion

An unrelated cast (or `void*` cast) type confusion occurs when an object is cast to an entirely unrelated type. It is only safe when the object being cast is an instance of the destination type. If the object being cast is not of the expected type, it can lead to arbitrary command execution.

Exploitation. Listing 2 shows an example of such an exploitation. Here, a `void` pointer is cast to a class `UnrelatedPrinter` and a method `print` is called on it. An exploit function creates an instance of `ChildExecutor` and passes it to the vulnerable function, which incorrectly assumes it to be an instance of `ChildPrinter` and hence calls the method `exec` instead of `print`.

```cpp
#include <iostream>
class UnrelatedPrinter {
public:
    virtual void print(const char *str) { std::cout << str << std::endl; }
};
class UnrelatedExecutor {
public:
    virtual void exec(const char *cmd) { system(cmd); }
};
void unrelatedCast(void *p, const char *msg) {
    ((UnrelatedPrinter *)p)->print(msg);
}
void exploit() {
    UnrelatedExecutor *p = new UnrelatedExecutor();
    unrelatedCast(p, "/bin/sh");
    delete p;
}
```

Listing 2: Example of an exploit of an unrelated cast type confusion

2.2 Type++ Dialect

Type++ [1] is a novel dialect of C++ designed to effectively mitigate type confusions.

In standard C++ runtime checks can only be performed on polymorphic classes, as they contain Run Time Type Information (RTTI). The absence of continuous type information throughout an object's lifetime presents a challenge in enforcing consistent runtime checks and preventing type confusion.

Type++ introduces a new paradigm where every allocated object carries type information throughout its entire lifespan. This design ensures that type checks can be applied whenever and wherever necessary, promoting program correctness. Unlike standard C++, which limits runtime verification to polymorphic classes, type++ enables dynamic verification for all casts.

The primary objective of type++ is to address type confusion bugs at the language level. While this entails sacrificing some compatibility with traditional C++, it ensures that all casts are checked at runtime, guaranteeing that objects are only within the bounds of their respective types as established during initialization.

2.2.1 Levels of Class Collection

To balance compatibility concerns, type++ offers three distinct levels of collection of classes to instrument:

1. *Explicit Runtime Types*: Associate at compile time a unique type T to each class A in a program P. This approach enables universal checks based on the explicit type of each live object. Consequently, all instances of all classes include a field that unambiguously identifies their type.
2. *Explicit Runtime Types for Cast Objects*: Associate at compile time a unique type T to each class *A involved in a casting operation* in a program P. This property ensures runtime verification for objects involved in cast operations.
3. *Explicit Runtime Types for Annotated Objects*: Associate at compile time a unique type T to each class *A annotated* in a program P. This level enables the application of type++ to complex programs, such as Chromium, while managing resource constraints effectively.

These levels of class collection in type++ provide flexibility, allowing developers to choose the appropriate level of runtime checks based on their specific requirements and trade-offs between compatibility and security.

2.3 Chromium Browser Overview

The Chromium browser [4] is an open-source web browser project that serves as the foundation for various popular browsers, including Google Chrome, Microsoft Edge and Opera. It is written primarily in C++ and uses the Blink rendering engine [3], which is responsible for parsing and rendering web content. It also includes a multi-process architecture [5], where different components run in separate processes, providing improved stability and security.

2.3.1 Architecture of Chromium

Chromium's multi-process architecture [5] consists of the following components:

- *Browser*: This is the main process that manages the user interface, handles user input and coordinates the activities of other processes.
- *Renderers*: By default, each tab in Chromium runs in its own renderer process, which is responsible for rendering web content using the Blink rendering engine and executing JavaScript

code. Each renderer process operates in a sandboxed environment to prevent malicious web content from affecting the system, e.g., by ensuring the renderer's network access is only through the network service, by restricting filesystem access with the host operating system's permissions and by limiting access to the user's display and input.

- *Additional processes*: Chromium has also other components in separate processes, such as the GPU process, the network service and the storage service.

The browser process and the renderer processes communicate using the Mojo Inter-Process Communication framework (see Subsection 2.3.2).

2.3.2 Mojo IPC Framework in Chromium

The Mojo Inter-Process Communication framework [15] is a collection of runtime libraries which abstract the communication between processes. It provides a C++-like interface definition language for defining interfaces for making inter-process calls. The interfaces can be translated into abstract classes in C++.

Mojo allows to exchange messages and data between processes while maintaining isolation and sandboxing. It is designed to overcome the limitations and security risks associated with traditional inter-process communication methods. It is a critical component of the Chromium browser, enabling communication between different processes within the browser's architecture.

Any type confusion bugs in Mojo could potentially be exploited to compromise the security of the browser, enabling unauthorized access or malicious actions.

2.3.3 Skia Graphics Library in Chromium

Skia [19] is an open-source 2D graphics library that provides a comprehensive set of APIs for rendering text, geometries and images. It is used as the graphics engine of Chromium and also other critical products, such as the web browser Mozilla Firefox and the operating systems Android and Chromium OS.

Skia offers a range of features, including anti-aliasing, blending, filtering and image decoding. Within Chromium, it works closely with the Blink rendering engine to render web pages, apply CSS styles, handle animations and compose visual elements.

This library is an interesting target of analysis due to its widespread usage.

2.4 Related Work

Several related works aim at handling possible type confusions:

- Other programming languages such as Java and Rust [14] were designed as type-safe from their conception.
- The C dialects Cyclone [12] and CCured [16] extend the language with a set of protections in order to prevent type confusions at compile time. They are incompatible with the C++ specification.
- The sanitizer Ironclad C++ [8] enforces type safety in C++ programs but does not handle unrelated casts.
- The sanitizers LLVM-CFI [13] and LLVM-UBSan [20] check for type confusions at runtime but perform the checks based on RTTI, thus work only with polymorphic classes.
- The sanitizer HexType [11] tracks all the objects and checks their types against disjoint metadata. However, it incurs high overhead and imprecision, e.g., if allocators are incorrectly tracked, resulting in false positives and false negatives.

The type++ compiler is designed for C++ code and prevents both down-to cast and unrelated cast type confusions. It is built on top of the infrastructure of Clang/LLVM version 13.0.0 and patches it to be able to emit warnings when encountering type confusions and to instrument the objects of non-polymorphic classes. It performs the type casting verification using part of LLVM-CFI [13]. Since type++ augments all classes with RTTI and LLVM-CFI performs type checks only over objects with RTTI, type++ extends the coverage of vanilla LLVM-CFI.

Chapter 3

Analysis

3.1 Integration of type++ in Chromium

Chromium frequently faces type confusion vulnerabilities [10] [17] [22] [7]. With over 35 MLoC in C++, it is one of the biggest active open-source C++ project.

Chromium developers opt not to utilize `dynamic_cast` in release builds for better performance, relying instead on the potentially unsafe `static_cast`. This compromise in performance carries a security drawback. Compiling Chromium with type++ would improve the browser security and serve as a benchmark to evaluate the real-world effectiveness of type++.

The integration of type++ in the Chromium browser involves modifying the existing codebase to incorporate its associated RTTI mechanisms. We analyze bug confusions reported in Chromium version 90.0.4430.212, distributed with Debian 10, which is used in the evaluation of type++ [1].

3.2 Trace of Type Confusion Bugs Reported by type++

We compiled Chromium with type++ and left it running the JavaScript benchmark suite JetStream 2 [2], saving into a file the trace of all type confusions identified by type++ during its execution. The trace showed 54 type confusions in Skia with the same source and destination types and 15 type confusions in Mojo.

We focused mainly on the type confusion bug in Skia due to its prevalence in the trace. Skia is widely used and has a significant impact on various software systems besides Chromium, making it crucial to understand and address such a vulnerability.

3.3 Type Confusion Edge Case in Skia

The analysis of the most reported type confusion bug in the type++ trace led to the discovery and fix of an edge case that was not handled properly by type++. It is triggered in two C-style unrelated casts:

1. One from the type `SkDQuad` to the type `SkDCurve` in `third_party/skia/src/pathops/SkDQuadLineIntersection.cpp:310`:

   ```
   double quadT = ((SkDCurve*) &fQuad)->nearPoint(SkPath::kQuad_Verb,
       (*fLine)[lIndex], (*fLine)[!lIndex]);
   ```

2. Another from the type `SkDConic` to the type `SkDCurve` in `third_party/skia/src/pathops/SkDConicLineIntersection.cpp:210`:

   ```
   double conicT = ((SkDCurve*) &fConic)->nearPoint(SkPath::kConic_Verb,
       (*fLine)[lIndex], (*fLine)[!lIndex]);
   ```

According to the message of the commit that introduced them [6], the methods "check to see if the end of the line nearly intersects the curve". The casts are triggered whenever the Status Bar in the bottom left of the Chromium GUI appears. Since the status bar appears spontaneously, the casts are reached after the startup of Chromium without the need of any manual intervention.

Listing 3 exemplifies the custom way of type identification used in Skia. It defines several structures, namely `Trio`, `Pair` and `PairContainer`. `Trio` contains an array of three elements, `Pair` contains an array of two elements and `PairContainer` contains an object of type `Pair` which, thanks to the overloading of the subscript operator, can be accessed as if it were itself an array of two elements.

An unrelated base structure `UnionBase` uses a union to hold an instance of the different structures. This, together with the use of a C-style cast, allows for polymorphic behavior. A C-style cast or more specifically a `reinterpret_cast` from one class A to an unrelated class B can be used to simulate inheritance when B includes as its first member a union with an object of class A. Developers of other projects, such as POV-Ray [18] and Xalan-C++ [21], have employed similar approaches. For the analysis of their implementations one can refer to type++'s article [1]. However, a cast between two unrelated structures is undefined behavior, since the compiler may optimize the memory layout of the structures, causing differences that would render such casts invalid and potentially exploitable.

A print function casts the specific type `PairContainer` to the unrelated base type `UnionBase` and calls `printValues` on it passing as argument the type identifier. This method prints the stored values based on the type. It invokes the function `typeToCount` to determine the number of values

to print based on the type identifier and accesses the values via the union interpreting the object as if it were a trio, i.e., it contained three values.

Before this analysis, in this scenario type++ collected `Pair` instead of `PairContainer` and thus instrumented the instances of `Pair` and ignored the instances of `PairContainer`. Moreover, it unnecessarily collected and instrumented the destination type `UnionBase`. As evident in the figure 3.1, before the fix the object `pair_container` was not instrumented and therefore contained the vtable pointer of `Pair` as first element in memory. For this reason, type++ mistakenly recognized instances of `PairContainer` as instances of `Pair` and reported the type confusion in the cast as having the source type `Pair` instead of `PairContainer`.

After the fix of the class collection, type++ correctly collects `PairContainer` and hence instruments the instances of `PairContainer` accordingly. As shown in the figure 3.1, now after the fix the object `pair_container` contains as first element the vtable pointer of `PairContainer`. As a result, when the print function casts the instance of `PairContainer` to `UnionBase` type++ correctly interprets the source type as `PairContainer`.

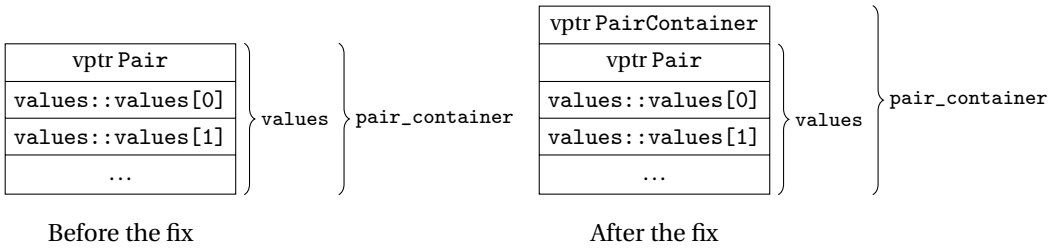

Figure 3.1: Comparison of the memory layout of an instance of `PairContainer` in memory before and after the fix of type++

Possible violation. In practice, Skia calls the method `addLineNearEndPoints` on an object of type `LineQuadraticIntersections`, which casts the property `fQuad` of type `SkDCubic` in structure `SkDQuad` to structure `SkDCurve` and calls the method `nearPoint` of `SkDCurve` on the cast object. `SkDCurve` contains as first property a union with an object of type `SkDQuad` and, among others, an object of type `SkDCubic`. `nearPoint` uses the union to interpret the object `fQuad` as if it were an object `fCubic` of type `SkDCubic` and accesses in a loop the elements of the property `fPts` of `fCubic`, an array of 4 elements, bigger than `fPts` of `SkDQuad`, which has 3 elements. The loop stops at index 2, returned from the call to `SkPathOpsVerbToPoints` with the object `verb` of enum `SkPath::Verb` as argument, which is passed by `LineQuadraticIntersections` and whose value is `SkPath::kQuad_Verb` (2). If there were a mismatch between the value `verb` of enum `SkPath::Verb` and the type `SkDQuad` of the original object, as type++ wrongly reported before the fix of the edge case, there would be a Spatial Memory Violation. We submitted a bug report describing the issue to the developers of Skia. Appendix A contains the Proof of Concept.

```cpp
#include <iostream>

struct Trio          { int values[3]; };
struct Pair          { int values[2];
                       int& operator[](size_t i) { return values[i]; } };
struct PairContainer { Pair values; };

enum Type {
    TYPE_TRIO,
    TYPE_PAIR,
    TYPE_PAIR_CONTAINER,
};
size_t typeToCount(Type type) {
    switch (type) {
        case TYPE_TRIO:           return 3;
        case TYPE_PAIR:           return 2;
        case TYPE_PAIR_CONTAINER: return 2;
    }
}

struct UnionBase {
    union {
        Trio          trio;
        Pair          pair;
        PairContainer pair_container;
    };
    void printValues(Type type) {
        for (size_t i = 0; i < typeToCount(type); i++)
            std::cout << trio.values[i] << std::endl;
    }
};

void printPairContainer(PairContainer *pair_container) {
    // type++ would collect Pair instead of PairContainer and also UnionBase
    ((UnionBase *)pair_container)->printValues(TYPE_PAIR_CONTAINER);
}
```

Listing 3: Example of the custom type identification in Skia

3.4 Type Confusions in Mojo

The other reported type confusions in the type++ trace are triggered in two casts:

1. The more common cast, reported 11 times, is an unrelated cast via `static_cast` from several types to the type `mojo::core::ports::(anonymous namespace)::UserMessageEventData` in `mojo/core/ports/event.cc:232`:

   ```
   auto* data = static_cast<UserMessageEventData*>(buffer);
   data->sequence_num = sequence_num_;
   ```

2. The less common cast, encountered 4 times, is an unrelated cast via `reinterpret_cast` from the type `base::ScopedGeneric<int, base::internal::ScopedFDCloseTraits>` to the type `mojo::core::BrokerMessageHeader` in `mojo/core/broker_messages.h:72`:

   ```
   BrokerMessageHeader* header =
       reinterpret_cast<BrokerMessageHeader*>(message->mutable_payload());
   ```

Type Confusions in `event.cc`

The type confusions in `event.cc` are triggered in the method `SerializeData` of the class `UserMessageEventData`. This method is called in the method `Serialize` inherited from the parent class `Event`.

In order of occurrences in the trace, the source types are:

- `mojo::InterfaceEndpointClient::HandleIncomingMessageThunk` (3 times)
- `mojo::core::WatcherDispatcher` (3 times)
- `viz::TileDrawQuad` (1 time)
- `mojo::core::MessagePipeDispatcher` (1 time)
- `mojo::internal::ControlMessageHandler` (1 time)
- `blink::mojom::BlobStub<mojo::RawPtrImplRefTraits<blink::mojom::Blob»` (1 time)
- `base::ScopedGeneric<int, base::internal::ScopedFDCloseTraits>` (1 time)

The destination type `UserMessageEventData` is a structure containing a sequence number, a number of ports and an integer representing a padding.

Type Confusions in `broker_messages.h`

The type confusions in `broker_messages.h` are triggered in the method `CreateBrokerMessage` of the class `Channel::MessagePtr`.

The source object is obtained from the method `mutable_payload` called on the object `message`. It is raw data addressed via a `void` pointer. Before the cast, the object `message` is only initialized. Its initialization is based on two parameters `message_size` and `num_handles`, which are the only values that can be possibly controlled.

The destination type `BrokerMessageHeader` is a structure containing a type identifier and an integer representing a padding.

Chapter 4

Conclusion

In this project we assessed the compatibility of the type++ dialect [1] with the Chromium browser [4], focusing on analyzing the most reported type confusions within the codebase. Type confusion bugs pose serious security risks, as they can allow attackers to manipulate objects and potentially compromise user data or execute malicious code [10] [17] [22] [7]. By compiling Chromium with type++ and running the JetStream 2 benchmark suite [2], we identified a total of 54 type confusions in the Skia graphical library [19] and 15 type confusions in the Mojo Inter-Process Communication framework [15].

Our analysis primarily focused on the type confusion bug in Skia due to its prevalence. Through this analysis, we discovered an edge case in type++ that was not handled properly. This edge case occurred in C-style unrelated casts and led to incorrect type identification by type++. It mistakenly collected the wrong classes and ignored the correct ones, resulting in inaccurate reporting of type confusions. By fixing the class collection mechanism in type++ we were able to address this edge case. After the fix, type++ correctly identified and instrumented the instances of the correct types, ensuring accurate detection of type confusions.

Additionally, we identified type confusions in Mojo triggered by two specific casts. Their existence demonstrates the relevance of addressing type confusions in complex software applications like Chromium. Further investigation is required to understand the impact of these type confusions.

In conclusion, the project highlights the potential of the type++ dialect to prevent type confusions and improve security in C++ programs. By identifying and addressing an edge case, we have contributed to enhancing the compatibility and effectiveness of type++ in real-world applications. Continued research in this area is crucial to improving the robustness of C++ programs and enhancing security in software systems.

Bibliography

[1] Nicolas Badoux and Mathias Payer. “type++: Prohibiting Type Confusion with Inline Type Information”. In: (2023).

[2] Saam Barati and Michael Saboff. *Introducing the JetStream 2 Benchmark Suite*. Mar. 27, 2019. URL: https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite/.

[3] *Blink (Rendering Engine)*. The Chromium Authors. URL: https://www.chromium.org/blink/.

[4] *Chromium*. The Chromium Authors. URL: https://www.chromium.org/Home/.

[5] *Chromium - Multi-process Architecture*. The Chromium Authors. URL: https://www.chromium.org/developers/design-documents/multi-process-architecture/.

[6] Cary Clark. *55888e44171ffd48b591d19256884a969fe4da17: pathops coincidence and security rewrite*. Git at Google. July 18, 2016. URL: https://skia.googlesource.com/skia/+/55888e44171ffd48b591d19256884a969fe4da17.

[7] *CVE-2022-3315: Type confusion in Blink*. Chrome Releases. May 5, 2022. URL: https://chromereleases.googleblog.com/2022/09/stable-channel-update-for-desktop_27.html.

[8] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M. K. Martin, and Steve Zdancewic. “Ironclad C++: a Library-Augmented Type-Safe Subset of C++”. In: *ACM SIGPLAN Notices* 48 (10 2013), pp. 287–304.

[9] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990, pp. 227–232. ISBN: 0-201-51459-1.

[10] Guang Gong and Qihoo 360 Alpha Team. *CVE-2017-5108: Type confusion in PDFium*. Chrome Releases. Feb. 24, 2017. URL: https://chromereleases.googleblog.com/2017/07/stable-channel-update-for-desktop.html.

[11] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. “HexType: Efficient Detection of Type Confusion Errors for C++”. In: *ACM Conference on Computer and Communication Security* (2017).

[12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. *Cyclone: a safe dialect of C*. 2002.

[13] *LLVM Control Flow Integrity - Bad Cast Checking*. The Clang Team. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html#bad-cast-checking.

[14] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *ACM SIGAda Ada Letters* 34 (3 Oct. 18, 2014), pp. 103–104.

[15] *Mojo docs (go/mojo-docs) - Mojo*. The Chromium Authors. URL: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/mojo/README.md.

[16] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. *CCured: type-safe retrofitting of legacy software*. May 1, 2005.

[17] Alexandru Pitis and Microsoft Browser Vulnerability Research. *CVE-2019-5757: Type Confusion in SVG*. Chrome Releases. Dec. 15, 2018. URL: https://chromereleases.googleblog.com/2019/01/stable-channel-update-for-desktop.html.

[18] *POV-Ray - The Persistence of Vision Raytracer*. Persistence of Vision Raytracer Pty. Ltd. URL: https://www.povray.org.

[19] *Skia: The 2D Graphics Library*. URL: https://skia.org.

[20] The Clang Team. "LLVM UndefinedBehaviorSanitizer". In: (). URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[21] *Xalan-C++ version 1.10*. The Apache Software Foundation. URL: https://xml.apache.org/xalan-c/.

[22] Looben Yang. *CVE-2020-6464: Type Confusion in Blink*. Chrome Releases. Apr. 15, 2020. URL: https://chromereleases.googleblog.com/2020/05/stable-channel-update-for-desktop.html.

Appendix A

Proof of Concept for the Type Confusion in Skia

To trigger the type confusion in Skia, it is sufficient to visit the following webpage as a Proof of Concept:

```
<!DOCTYPE html>
<html lang="en-US">

<head>
    <meta charset="utf-8">
    <title>Status Bar PoC</title>
    <script>
        window.addEventListener("load", () => {
            document.getElementById("anchor").focus()
        })
    </script>
</head>

<body>
    <a id="anchor" href="#"></a>
</body>

</html>
```