École Polytechnique Fédérale de Lausanne

Building a robust test suite of type confusion vulnerabilities

by Adrien Bouquet

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

The External Reviewer
External Expert

Nicolas Badoux
Thesis Supervisor

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

June 7, 2024

"Before I came here I was confused about this subject. Having listened to your lecture I am still confused. But on a higher level."

- Enrico Fermi

Acknowledgments

I would like to thank my supervisor, Nicolas Badoux, for his guidance and support throughout this project. Finally, I would like to thank my family and friends for their support

Lausanne, June 7, 2024 Adrien Bouquet

Abstract

C++ has a lot of vulnerabilities due to type confusion, which can lead to memory corruption and security vulnerabilities. Casting a pointer from a type to another is a highly common operation but, if misused, can lead to type confusion and thus can be exploited. Hextype[4] and Type++[1] are two mechanisms that aim to detect type confusion at runtime. Although they achieve a high success rate, they can not ensure a perfect type safety.
We propose a benchmark that evaluates a range of type confusion cases by defining which situations are likely to trigger type confusion. We evaluate it on Hextype and Type++ and we conclude that they achieve a high success rate even if they do not pass all the tests. In addition, we show that Type++ is more complete than Hextype as it detects more type confusion.

Contents

Chapter 1

Introduction

According to *The Importance of Being Earnest* (TIOBE) index, C++ is the 3rd most used programming language in the world [9], exceeding even Java. Both languages have in common object-oriented programming, a paradigm that allows to organize code in objects that contain both data and functions. In C++, objects are organized in hierarchies, from base classes to derived classes and derived classes inherit the characteristics of their base classes. However, this feature can lead to type confusion vulnerabilities. A type confusion is a vulnerability that occurs when a pointer is cast from a type to another type which is not compatible. Hence, the compiler misinterprets the pointer. This mistyped pointer can be then exploited by an adversary to execute arbitrary code, heap corruption and hijack the control-flow. For instance, in 2021, a type confusion vulnerability was discovered in Telegram[6] and could allow an adversary to access the heap memory.
To prevent these vulnerabilities, sanitizers such as Hextype[4] and type++[1] have been developed. These tools aim to detect type confusions at runtime. Even if they achieve a high success rate, they can not ensure a perfect type safety, and we want to measure their completeness.
In this paper, we propose a benchmark based on multiple perspectives of type confusion vulnerabilities. We present the concept of initialization, modification and deletion of objects, and some special cases that are worthwhile to evaluate due to their complexity. We evaluate this benchmark on Hextype and type++ and we conclude that they achieve a high success rate even if they do not pass all the tests.

Chapter 2

Background

C++ is a general purpose programming language widely-used in a lot of domains such as operating systems, or machine learning. As C++ is a compiled language, it allows developers to write extremely fast and efficient code that run with minimal overhead.
Types are a core concept in C++ but typecasting, which is the action of casting from a type to another, is source of vulnerabilities. For instance, type confusion is a common security issue in C++, which occurs when the compiler misinterprets a pointer type generally after a wrong typecasting.
This can lead to memory corruption and security vulnerabilities. An adversary can then exploit this vulnerability to execute arbitrary code, heap corruption and hijack the control-flow. Such an attack occurred in the past in V8 in Google Chrome, where it allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page (CVE-2023-3079) [8]. Hence, it is crucial to detect wrong typecasting which induce type confusion. These wrong casts can occur when casting a pointer outside a type hierarchy (`char*` to `void*` then casting to another type, see line 40 in Listing 2.1) or inside a type hierarchy with classes and object-oriented programming e.g., casting a pointer from a base class to a derived class. This can lead to memory corruption and security vulnerabilities.
In this chapter, we will present the concept of the object-oriented programming, class hierarchies and inheritance in C++ section 2.1, and present the different type casting operators in C++ section 2.2. Finally, we will present the different mitigation mechanisms to type confusion and how they handle it section 2.3.

2.1 Object-Oriented Programming, Class Hierarchies and Inheritance

Object-Oriented Programming (OOP) is ǫncept focus on creating objects which are instances of classes that contain both data and functions. This paradigm provides data abstraction, faster execution and code reusability. Classes are organized in hierarchies, generally with one base class and one derived, and can inherit attributes and methods from other classes which they then can

```cpp
#include <string>
#include <iostream>
using namespace std;

class Base {}; // Base class
class Dummy: public Base { // Derived class
	public:
	virtual void printMessage(string message) {
		cout << message << endl;
	}

};

class Password: public Base { // Derived class 2
	string password = "empty";
	// private function that cannot be accessed from outside the class
	virtual void modifyPassword(string newPassword) {
		password = newPassword;
	}
	public:
	Password(string newPassword) { modifyPassword(newPassword); }

	void printPassword() { cout << password << endl; }
};

class Unrelated { // Unrelated class
	public:
	virtual void doRandomStuffWithString(string str) {
		/* random stuff*/
		cout << str << endl;
	}
};

void typeConfusionInHierarchy(Base* base) { // Type confusion inside the hierarchy
	Dummy* dummy = static_cast<Dummy*>(base);
	dummy->printMessage("You have been hacked"); // Illegal access
}

void typeConfusionOutsideHierarchy(Base* base) { // Type confusion outside the hierarchy
	Unrelated* unrelated = reinterpret_cast<Unrelated*>(base);
	unrelated->doRandomStuffWithString("You have been hacked twice"); // Illegal access
}

int main() {
	Password password("MyHardPassword"); // Password initialization
	Base* base = &password; // upcast to base class
	password.printPassword(); // print "MyHardPassword"
	typeConfusionInHierarchy(base); // polymorphism
	password.printPassword(); // print "You have been hacked"
	typeConfusionOutsideHierarchy(&password);
	password.printPassword(); // print "You have been hacked twice"
	return 0;
}
```

Listing 2.1: Example of type confusion in C++ within the same class hierarchy.

use or override. In Listing 2.1, classes `Password` and `Dummy` are derived from the base class `Base`, whereas the class `Unrelated` is unrelated to their class hierarchy. A derived class can have multiple base classes and thus possesses all the methods and attributes from its base classes. Furthermore, a derived class is a subtype of its base class, meaning that a derived class can be used in place of its base class e.g., at line 48 a `Password` pointer is given instead of a `Base` pointer. This concept which is called polymorphism, is remarkably practical but leads to some issues. For instance, when a base class has two derived classes that are not directly related to each other, it is possible to cast a pointer from one derived class to another in type casting by the base class. This comes from C compatibility as all casts are allowed in C and C++ developers tried to keep that. However, this behavior induces type confusion, as at line 35, where a `Password*` is cast to a `Dummy*` after being cast to a `Base*`.

2.2 Type Casting

Typecasting is the operation of casting a pointer from a type to another. C++ provides four casting operators: `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`.

`const_cast<target-type>(expression)` revokes the `const` or `volatile` attribute of an object. Due to operator's behavior, it does not impact type hierarchies and thus does not lead to type confusion, although there are some issues with `const_cast`. It is out of scope of this paper and will not be discussed further.

`static_cast<target-type>(expression)` cast an expression to a specified type while checking at compile-time if target type and expression's type are related but not at runtime. Therefore, this reduces cast verification to checking the types in the type hierarchy. It does not guarantee the types are compatible at runtime and places the onus back on programmers to ensure type correctness. Hence, it leads to a type confusion vulnerability when misused i.e., casting from the base class to a derived class or when casting a pointer to a void pointer.

`reinterpret_cast<target-type>(expression)` similarly to static_cast, it casts an expression to a specified type by reinterpreting the underlying bit pattern but with the difference that it does not check anything neither at compile time nor at runtime. Hence, it can potentially cast two incompatible types and lead to type confusion. It is also the programmer's responsibility to ensure the correctness of the target-type and the type of expression.

`dynamic_cast<target-type>(expression)` cast safely an expression to a specified type and checks at compile-time and at runtime, unlike `static_cast`, if the types are compatible. To achieve the runtime check, some forms of `dynamic_cast` rely on run-time type identification (RTTI), that is, information about each polymorphic class in the compiled program. However, this metadata is only available when the class has at least one virtual function and thus is polymorphic, which constraint to use `dynamic_cast` only on polymorphic classes.

Finally, C-style typecasting exists in C++ due to backward compatibility between both languages but highly discouraged due for instance to its lack of clarity for programmers. Furthermore, the compiler translate C-style typecasting to (i) `const_cast`, (ii) `static_cast` and (iii) `reinterpret_cast` in this preferred order which passes on the underlying types confusions.

2.3 Type confusions and Mechanisms

As previously mentioned, type confusion vulnerabilities can lead to some memory corruption and security vulnerabilities which products such as Telegram, (CVE-2021-31318[6], CVE-2021-31317[5]) or Google Chromium (CVE-2022-2295[7]) are exposed to. Listing 2.1 shows an example of a type confusion. First, we define a base class `Base` and two derived classes `Dummy` and `Password` with their own attributes and methods. We set up the password to "MyHardPassword" (line 45) of our class. A type confusion inside a type hierarchy can occur when there is a cast from a base to a derived class. In our example, the password pointer is cast to its base and passed (line 48) to `typeConfusionInHierarchy()` (line 36) which cast it to a `Dummy*` (line 35). From this line, the behavior is undefined and can lead to memory corruption. Even if it's calling `printMessage()` of `Dummy`, it will execute `modifyPassword()` which will modify our password. This behavior is due to object being a `Password` object, and it will call `modifyPassword()` method. A type confusion outside a type hierarchy can occur when there is a cast from a base to an unrelated class. We can observe this behavior in `typeConfusionOutsideHierarchy()` (line 41) where the password pointer is cast to an `Unrelated*` (line 40). Calling `doRandomStuffWithString()` will call in fact `modifyPassword()` and modify our password a second time. An adversary could exploit vulnerabilities, especially if the program takes some inputs to the program from its user. However, some downcast or unrelated cast be legitimate when the programmer is sure of the type of the object, and we want to avoid false positive as well as false negative.

Hence, it is a problem that require to be handled for developer and user safety. The compiler checks statically the correctness of each cast to ensure that the types are compatible, but it does not guarantee the same correctness at runtime, because for instance an object of a derived class can be used in place of an object of a base class due to polymorphism and vice versa as in line 48. To detect type confusion at runtime, there exists two approaches: (i) relying on disjoint metadata, such as type table to keep track of each type of each object or trees for type hierarchy, and (ii) relying on existing metadata Runtime Type Information (RTTI), such as vtables. Generally, the first approach is more complete because it covers more types, but it introduces overhead and performance issues. The second is faster but less complete due to its limitation to polymorphic classes, and it caused some crash issue. In this paper, we will focus and compare on two methods: Hextype [4] and type++[1]. Both checks types correctness at runtime, but they differ in their approach. Hextype is older and relies on disjoint metadata to achieve types correctness. Type++ is a recent C++ dialect and relies on existing metadata to achieve types correctness but unlike an approach based solely

on existing metadata for polymorphism object, it extends this by adding a vtable pointer to each object to increase efficiency, cover more types and avoid crash issue. In the following, we will detail how Hextype and type++ work. Comparing them offers a perspective on the effectiveness of each approach, despite the age difference between the development of these two mechanisms.

2.3.1 Hextype

HexType is a Clang/LLVM-based type confusion sanitizer used to detect type confusion vulnerabilities in C++ programs. It relies on disjoint metadata to keep track of each type of each object and to check types correctness at runtime. When compiling a program, Hextype creates a type table which contains all the relationship information. Then at runtime, it collects the types of each allocated object inside called object mapping table. Finally, when a cast happens, it verifies if the cast is correct by comparing the two previously defined tables and thus detect type confusion. However, Hextype has some limitations, as it considers only type confusion between derived classes and not between unrelated classes.

2.3.2 type++

Type++ is a recent C++ dialect that extends the C++ language to prevent type confusions. It relies on existing metadata, more precisely vtable pointer, to achieve types correctness, but instead of keeping only polymorphic objects, it extends this property to all objects. At compile, it adds that vtable pointer to each object to keep track of the type of the object and then at runtime, it checks if the cast is correct by comparing the vtable pointer of the object.

Chapter 3

Design

Type confusion sanitizers are designed to detect type confusion vulnerabilities in C++ programs. Our threat model assumes that an adversary can exploit type confusion vulnerabilities to execute arbitrary code. We built a test suite with a multitude of cases to evaluate how a sanitizer or a C++ dialect can detect these types confusion and, a fortiori, try to prevent such adversaries from exploiting these vulnerabilities, In this chapter, we will present the design of the test suite for type confusion vulnerabilities, which decisions or language specifications we want to evaluate the sanitizer on.

3.1 Benchmarking Type Confusion

3.1.1 Initialization

Initialization is a highly important step in a program. When creating an object, its type is assigned. At this point, both methods act on values to keep track of their types, type++ by adding a vtable pointer to every object and Hextype by recording all necessary information in its disjoint metadata. We will evaluate their completeness i.e., their capacity to keep track and detect type confusion over a maximum of initialization methods. In the following, we will present various specific forms of object initialization on which we will evaluate the sanitizers.

Direct initialization

Direct initialization is the simplest way to initialize an object and thus essential to measure if sanitizers efficiency. It consists of a creating an object from a set of arguments. In fact in Listing 2.1, `password` is directly initialized with the string "MyHardPassword" at line 45.

Copy initialization

Copy initialization is another way to initialize an object. It consists of creating an object from another object. With Copy initialization, we want to ensure that a sanitizer keep track of the type of object when it is copied. For instance, if we copy an object of a derived class, we expect methods to keep track of the type of this object and not lose any information, especially when casting to another type.

List initialization

List initialization, directly or copy initialized, is a way to initialize an object from a list of arguments. It is a more recent way as it appears with C++11 to initialize an object and is particularly worthwhile to evaluate the completeness of the sanitizer.

C-style Allocators and `new` operator

C-style allocators such as `malloc`, `calloc` and `realloc` are used to allocate memory. They are a heritage from C and are still used in C++. The `new` operator is a C++ operator that allocates memory and initializes the object. These operators allocate memory dynamically i.e., at runtime and without a fixed size. Unlike other methods which a return an object of a known type, they return a void pointer that can be cast to any type. Depending on the size of the memory allocated, it can contain multiple objects. Thus, it is interesting to observe how the sanitizer behaves when the memory is allocated with these functions and if they manage to keep track correctly of the type of the objects.

3.1.2 Modification & Deletion

Many edge cases happen when modifying the pointers by casting, modifying the object referenced to the pointer, or deleting a pointer and reusing it to observe or in order to study the limitations of sanitizers. For instance, `placement new` is a variation of `new` operator and allows the programmer to construct an object at already allocated memory. Within a same memory block, we can construct multiple objects of different types, which can lead to type confusion if these objects are not correctly initialized. Furthermore, we try to modify the pointers by casting, modifying the object referenced to, or deleting a pointer and reusing it to observe if it correctly detects type confusion. These examples are particularly known to be an issue for Hextype.

3.1.3 Special cases

Along with the initialization and modification of objects, there are some more precise cases that are interesting to evaluate because they are tricky to handle for a sanitizer. Next, we present some of these cases that generates type confusion vulnerabilities:

Multiple Inheritance

Multiple inheritance is a feature of C++ that allows a class to inherit attributes and methods from

```
1  class Base {};
2
3  class Derived : public Base {float x = 1.0;};
4
5  class Unrelated { public: void doSomething() {}};
6
7  union Union {
8      Base *b;
9      Unrelated *u;
10 };
11
12 int main () {
13     Union u;
14     u.b = new Base();
15     Derived* d = static_cast<Derived*>(u.b); // Type confusion
16     u.u->doSomething(); // Undefined behavior
17     return 0;
18 }
```

Listing 3.1: Union example

multiple classes. It allows the creation of complex class hierarchies and does not restrict the programmer to a single inheritance per object. Furthermore, its application is truly advantageous when objects have to share multiple characteristics of several type hierarchies.

Union

A union is a special class type that allows to store different types of data in the same memory location The feature is particularly useful for saving memory as only the last object assigned to it is actually stored, but can be risky to use and understand. When dealing with unions from the point of view of type confusion, we must not confuse type confusion with undefined behavior. In fact, it is up to the programmer to ensure that the type of the value retrieve from the union is the correct one. With union type confusions happen when the value retrieved is correct but is wrongly cast to another type. For instance in Listing 3.1, the pointer `u.b` references a `Base` object but is cast to a `Derived` object at line 15. This is a real type confusion and should be detected by the sanitizer. However, at line 16, the pointer `u.u` references the same object as `u.b`(due to union nature) but is cast to an `Unrelated` object. Trying to access `u.u` will lead to undefined behavior but is not a type confusion. Nevertheless, there exist tagged unions, also known as `variant`, from C++17 onwards, which are a safer union that uses a tag to keep track of the type of the object stored in the union. It will not be evaluated in this paper but could be interesting to evaluate in the future.

Phantom Casting

Phantom casting occurs when a base class and a derived have the same data layout. In this case, it is usually possible to cast a pointer from a base class to a derived class even if the object is not of the derived class which should be classified as a type confusion. This special case exists because of

```
1 class Base {};
2
3 class Derived : public Base {};
4
5 int main(){
6         Base* base = new Base();
7         Derived* derived = static_cast<Derived*>(base); // Phantom casting
8         return 0;
9 }
```

Listing 3.2: Phantom casting example

backward compatibility of C++ with C. Listing 3.2 shows an example of phantom casting at line 7. The pointer `base` references a `Base` object but is cast to a `Derived` object, but this is allowed.

Chapter 4

Implementation

We want to compare the results obtained from Hextype and type++ between them. The benchmark quality resides in the diversity and coverage offered by the tests. In the following we will concentrate on C++11, although there are more recent versions with more features dedicated to OOP. In addition, we mainly test the standard C++11 without any external libraries or frameworks to simplify and evaluate the basis of the language itself. Next, we would describe some challenge we faced during the implementation of the test suite.

Set up sanitizers

Setting up sanitizers was the first step in the implementation of the test suite. The test suite is designed to be run with Hextype and type++ to evaluate their completeness on type confusion vulnerabilities. It is located inside type++ architecture and did not need to be installed. However, Hextype had to be installed and configured to be able to run the test suite. Hextype can only run up to C++14 and is not compatible with C++17 or C++20, which is a limitation for the test suite. Hextype only triggers derived type confusion whereas type++ triggers derived and unrelated type confusion inside and outside. Furthermore, Hextype should update its type table to reflect the new type of the object, when a pointer is modified or used after being deleted, but sometimes it fails to do so and produces false negatives or false positives.

Correctly trigger type confusion

One challenge was to correctly create our test cases such that we create legitimate type confusion. Unions, phantom casting and multiple inheritance were difficult to implement so that they generate legitimate type confusion and not undefined behavior or valid cast. For instance, making simple test cases by having very simple base class and a derived class could trigger phantom casting and hence no type confusion were spotted. On another hand, we spent a lot of time doing test cases on unions before realizing how to properly create confused types and not undefined behaviors. In order to reproduce the test suite, it is important to have a class structure that is not too complex

```
class A{};

class B: public A{
    int b = 3;
};

class C: public B{
    double c = 4.5;
};

int main(){
    A* a = new A();
    A* abis = new B();
    A* ac = new C();

    B* b = static_cast<B*>(a); // This is not allowed
    B* b2 = static_cast<B*>(abis); // This is allowed
    B* b3 = static_cast<B*>(ac); // This is allowed

    C* c = static_cast<C*>(a); // This is not allowed
    C* c2 = static_cast<C*>(abis); // This is not allowed
    C* c3 = static_cast<C*>(ac); // This is allowed
    C* c4 = static_cast<C*>(b2); // This is not allowed
    C* c5 = static_cast<C*>(b3); // This is allowed
}
```

Listing 4.1: Example of test cases (taken from derived_simple_cast.cpp)

but not too simple either, such that it does not trigger phantom casting or multiple inheritance and treating those cases separately.

Chapter 5

Evaluation

We first evaluate each mechanism separately. Subsequently, we will compare them, discuss the results and the limitations of the test suite.

Type++

Over the 38 test cases we created, type++ passes 35 and fails 3 of them. 2 of the failed due to multiple inheritanc, which make 92i̇% of success rate. The main issue is that type++ does not handle multiple inheritance. The test `multiple_parent`, a simple test case with a class that inherits from two classes. When we create an object of the second base class and cast it to the derived class, it should be detected as a type confusion. However, type++ does not detect it and classifies it as "missed". The last test case that failed is `union_casting`, where we try to cast a pointer on a union to a pointer of a value inside the union. This test case is not detected as a type confusion by type++.

Hextype

Over the 38 test cases we created, Hextype passes 31 and fails 7 of them which makes 81ḃ% of success rate. The main reason that justifies these failures is that Hextype does not handle very well C-style allocators. 4 test fail because of them. As we mentioned previously, these allocators are known to be a challenge for Hextype and its misses when we initialize object inside allocated memory. Regarding the last 3 failed test cases, 2 are caused by vectors and the last one is due to unions.

Discussion

When comparing the two mechanisms numerically, we can see that type++ is more complete than Hextype. Indeed, type++ detects more derived type confusion than Hextype. Another advantage of type++ is that it detects unrelated type confusion, which Hextype does not. Any idea to increase the completeness of Hextype would be to enforce the type table on all objects and not only on allocated object.

The objective of the test suite is to evaluate the mechanisms on a range of cases, and to trigger some edge cases that are known to be difficult to handle for sanitizers. However, it is important to highlight that our test suite has been made manually and is not exhaustive. Furthermore, the benchmark is restricted to standard C++11 does not take into account from newer versions. It does not include external libraries or frameworks to simplify the evaluation of the mechanisms, but they could be interesting to evaluate in the future. Finally, we could also test other sanitizers to compare them with Hextype and type++ and evaluate the completeness of our test suite.

Chapter 6

Related Work

Type confusions continue to be studied and Hextype and type++ are not the only mechanisms that aim to detect them. TypeSan [3] is a type confusion detector also based on LLVM framework. Similarly to Hextype, TypeSan relies on disjoint metadata to keep track of each type of each object and to check types correctness at runtime. However, TypeSan uses shadow memory scheme.
More recently, we can mention TCD [2] a static type confusion detector. Unlike the 3 mentioned above, it works by static analysis and is proven to handle complex feature as multiple inheritance or placement new.

Chapter 7

Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.

In conclusion, we have built a test suite based on C++ features and edge cases generating type confusion vulnerabilities. We have evaluated the completeness of Hextype and type++ on type confusion vulnerabilities using our benchmark. We have shown that type++ is more complete than Hextype as it detects more type confusion. Type++ achieve a 92i̇% success rate whereas Hextype achieves an 81ḃ% success rate. We pointed out that Hextype does not handle well C-style allocators, that type++ does not handle multiple inheritance, and they will serve as a basis for future work.

Bibliography

[1] Nicolas Badoux. *type++: Prohibiting Type Confusion with Inline Type Information.*

[2] *TCD: Statically Detecting Type Confusion Errors in C++ Programs.* 2019. URL: https://ieeexplore.ieee.org/abstract/document/8987463/figures#figures.

[3] *TypeSan: Practical Type Confusion Detection.* 2016. URL: https://dl.acm.org/doi/abs/10.1145/2976749.2978405.

[4] *HexType: Efficient Detection of Type Confusion Errors for C++.* 2024. URL: https://nebelwelt.net/files/17CCS.pdf.

[5] National Institute of Standards and Technology. *NIST CVE-2021-31317 Detail Page.* 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-31317.

[6] National Institute of Standards and Technology. *NIST CVE-2021-31318 Detail Page.* 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-31318.

[7] National Institute of Standards and Technology. *NIST CVE-2022-2295 Detail Page.* 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2295.

[8] National Institute of Standards and Technology. *NIST CVE-2023-3079 Detail Page.* 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2023-3079.

[9] *TIOBE index of june 2024.* 2024. URL: https://www.tiobe.com/tiobe-index/ (visited on 06/07/2024).