Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Compressing Type Information in Modern C++ Programs using Type Isolation

Masterarbeit von

Niklas Baumstark

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Simon Bischof, M.Sc.
	Andreas Fried, M.Sc.
Abgabedatum:	8. April 2019

Abstract

Modern memory corruption exploits targeting C++ programs often use the concept of forging C++ objects on the heap, and injecting them into the program by fully or partially corrupting pointers, or overwriting existing objects with the forged data. This technique has been used in past exploits to construct increasingly powerful primitives out of limited memory corruption bugs, to leak secret values used by mitigations such as *address space layout randomization* as well as to bypass *control flow integrity* mitigations. In order to make such exploits harder in practice, we propose to store out-of-band type information about C++ object types allocated on the heap. We compress this information and avoid performing expensive maintenance operations by using type-isolated heaps with useful regularity properties. Pointers into instrumented heap regions can be type-checked by performing at most two memory accesses.

To show the practicality of our approach, we describe a semi-automated process of adding our data structure to an existing code base with the help of a Clang frontend plugin. We successfully applied this process to the WebCore library, the main parsing and layout engine used by WebKit, which has been subject to various publicly disclosed memory corruption vulnerabilities in the recent past. In addition, we provide a plugin to automatically add type checks to a large portion of pointers fetched from the heap and subsequently used by the program.

Our evaluation concludes that while further optimizations would be required to reduce the performance overhead induced by our program transformation, a large subset of exploitable issues reported in 2017 and 2018 in the WebCore library is affected fundamentally by our instrumentation, in some cases rendering issues for which public exploits exist fully unexploitable, and in other cases significantly weakening the available exploit primitives.

Moderne Schadprogramme, die Sicherheitslücken im Zusammenhang mit Speicherbehandlung in C++-Programmen ausnutzen – sogenannte *Memory-Corruption-Exploits* – setzen häufig Techniken ein, welche darauf basieren, vom Angreifer kontrollierte Daten auf dem Heap als andere Datenstrukturen zu interpretieren. Dies kann gelingen, indem Zeiger manipuliert oder bestehende Zeiger, die noch auf alte Speicherbereiche zeigen, fälschlicherweise wiederverwendet werden. Auf diese Weise können komplexere, aufeinander aufbauende Primitiven wie relative und letztendlich absolute Lese- und Schreiboperationen konstruiert werden. Diese sind unter anderem notwendig, um explizit aus Sicherheitsgründen eingebaute Maßnahmen wie *Address Space Layout Randomization* und *Control Flow Integrity* zu umgehen. Um diese Technik der gezielten Typverwechslung deutlich schwieriger zu gestalten, schlagen wir eine Datenstruktur vor, die Typinformationen für Heapobjekte verwaltet. Die Effizienz dieser Datenstruktur ergibt sich aus einer Kompressionstechnik basierend auf typisolierten Heaps. Zeiger auf Objekte, welche von der Datenstruktur verwaltet werden, können mit höchstens zwei Speicherzugriffen auf ihren Typ hin geprüft werden.

Um die Praxistauglichkeit unseres Verfahrens zu zeigen, beschreiben wir einen semi-automatisierten Prozess, der sich eines Clang Frontend-Plugins bedient und sich auf ein beliebiges C++-Projekt anwenden lässt. Wir haben diesen Prozess erfolgreich auf die WebCore-Bibliothek angewandt, die für Dokumentenparsing und Layout in der WebKit-Browserengine zuständig ist und in der Vergangenheit vielen Sicherheitsproblemen unterlag. Zusätzlich stellen wir eine Instrumentierung ebenfalls auf Basis von Clang bereit, mit der Typchecks für vom Heap stammende Zeiger automatisch im Programm eingefügt werden können.

Unsere experimentelle Auswertung zeigt, dass viele der ausnutzbaren Sicherheitslücken, die in 2017 und 2018 in WebCore gefunden wurden, durch unsere Instrumentierung entweder nicht mehr ausnutzbar oder zumindest deutlich schwieriger in nützliche Exploitprimitiven umwandelbar sind. Im Gegenzug sind Performance-Einbußen zu erwarten, denen in Zukunft mit weiterer Optimierungsarbeit und statischer Analyse entgegengewirkt werden kann.

Contents

1. Introduction

State-of-the-art exploits for C++ programs often employ a multi-stage approach: In a first step, a limited memory corruption vulnerability such as a heap-based buffer overflow, use-after-free or type confusion is used to corrupt a C++ object on the heap. In one or multiple follow-up steps, the so far limited control over the target program's address space is increased by corrupting pointers, length fields, or indexes, thereby causing type confusions and out-of-bounds accesses relative to other objects. The result is full control over a pointer which is used to read and write attacker-controlled data, in a repeatable fashion. From here an attacker has multiple options. She might now try to

- exfiltrate private data already available in the process memory via their read primitive;
- modify data structures inside the program to escalate privileges, such as by disabling the same-origin policy in a web browser, or enabling deprecated plugin mechanisms such as ActiveX or Silverlight;
- corrupt data that influences the program's control flow directly, such as return addresses located on the stack, or function/vtable pointers stored in the heap.

Memory corruption mitigations typically focus on preventing control flow hijacks, but do not address the range of possible attacks that do not require arbitrary code execution. Furthermore, control flow hijacks are hard to mitigate using software-only defenses. As a practical example, Microsoft retired its *Return Flow Guard* technology, which is a state-of-the-art software shadow stack, because it could be broken completely by their own engineers in a scenario where the attacker can read and write arbitrary memory at arbitrary times [1]. Without RFG – which is now disabled in their products – their implementation of control-flow integrity (Control Flow Guard) can be easily bypassed by overwriting return addresses on the stack and using return-oriented programming. This will likely continue to be the case until hardware support for return address protection (e.g. via Intel CET or pointer authentication) is widely shipped to end users.

When comparing examples of recent exploits for web browsers and the Windows operating system, we can notice that a common demoninator is the careful crafting of counterfeit objects on the heap or kernel pool, which are then manipulated using the provided programming interfaces such as JavaScript APIs or system calls. A technique based on virtual function calls that makes heavy use of crafted objects was published as *counterfeit object-oriented programming* (COOP) [2]. We conjecture

that verifying the types of objects at runtime, using information provided by the heap allocator, can limit the usefulness of techniques based on crafted objects.

Our goal is to design a vulnerability-agnostic security mitigation that takes effect early in the exploitation process. It assumes a heap-based memory corruption vulnerability, and is supposed to make it considerably harder to escalate the available control over the program's memory and eventually, its control flow.

2. Preliminaries and Related Work

In this work we focus on modern, hardened software written in C++ such as web browsers. Based on a brief analysis of public vulnerabilities in the WebKit library – which we will elaborate on in chapter 4 – we are primarily concerned with heap memory safety rather than stack or global memory, since these issues constitute the vast majority of known vulnerabilities in the recent past.

2.1. Classifying memory safety

We can broadly categorize memory safety violations into the following categories:

Spatial memory safety The property that code operating on a complex heap object – such as a data buffer or array of objects – only refers to this object by performing memory accesses within the bounds of the corresponding heap allocation. The precise nature of what it means to *refer* to an object is defined by programming language semantics, which can become arbitrarily complex. A canonical example for C++ would be addressing an array's member via indexing or pointer arithmetics. A violation of this property implies the existence of a so-called *out-of-bounds access* vulnerability. These issues can be further categorized by whether the violating memory access is a load or store, and whether it is directly adjacent to the buffer. The case of a load or store directly after the heap object is referred to as a *heap-based buffer over-read/overflow*.

Temporal memory safety The property that code only refers to *live* heap objects, which are objects that have been allocated and not yet been deallocated. A precise definition can be given as follows: Given a heap allocator which never reuses memory and instead assigns a new address to each allocation, and also blacklists every byte of deallocated memory, the code never performs a memory access to blacklisted bytes. A violation of this property, which occurs as soon as a *dangling pointer* to freed memory is re-used, implies the existence of so-called *use-after-free* vulnerability. A common special case is a *double free* condition.

Type safety In a strongly typed setting, the property that a pointer of type T at all times points to some object of type U, where U is a subtype of T. The semantics of the low-level memory representations depend on the specific compiler implementation. A violation of this property is usually referred to as a *type confusion* vulnerability and can for example occur as the result of an unchecked downcast.

2.2. A simplified model of C++ types, objects and allocations

In order to reason about C++ programs, we introduce a simplified model of C++ types. Let T be the set of C++ types present in a program, which includes both primitive types and composite types defined via the **`class`** or **`struct`** keyword. We ignore cv qualifiers (**`const`**, **`volatile`**, **`mutable`** etc.) in this consideration. For a given pair of types $t, s \in T$, we define $\textsc{Offset}(t, s) \subseteq \{0, \ldots, \text{sizeof}(t) - 1\}$ as follows: Let x be the memory location of an object of type t. Then $\Delta \in \textsc{Offset}(t, s)$ if and only if the value $x + \Delta$ can be treated safely as a pointer to an object of type s. To give an example, consider the C++ class hierarchy and byte-level layout depicted below:

```
struct A {
  int x1;      // offset 0
};

struct B
  : public A   // base class at offset 0
{
  A x2;        // offset 4
  int x3;      // offset 8
  union {      // offset 12
    A x4;
    char x5;
  } u;
};
```

Here we have:

- $\textsc{Offset}(\texttt{A}, \textbf{\texttt{int}}) = \{0\}$
- $\textsc{Offset}(\texttt{A}, \texttt{A}) = \{0\}$
- $\textsc{Offset}(\texttt{B}, \textbf{\texttt{int}}) = \{0, 4, 8, 12\}$
- $\textsc{Offset}(\texttt{B}, \textbf{\texttt{char}}) = \{12\}$
- $\textsc{Offset}(\texttt{B}, \texttt{A}) = \{0, 4, 12\}$

This yields a basic characterization of type composition. For $t, s \in T$ with $\textsc{Offset}(t, s) \neq \emptyset$ we call s a *sub-record* of t and t a *super-record* of s. The set of sub-records and super-records of a type t are denoted as $\textsc{Sub}(t)$ and $\textsc{Super}(t)$, respectively. Note that the precise C++ type aliasing rules are rather complex, which is why in order to compute Offset in practice, we make use of certain heuristics and approximations.

To reason about object lifetime, we introduce the process of *object construction*: First, for some type t, **`sizeof`**(t) bytes of memory are allocated at some address x, then the type constructor C_t is called with x as the first argument. The object

ceases to exist after the type destructor D_t is called with x as the first argument. In between the calls $C_t(x, \ldots)$ and $D_t(x, \ldots)$, we say there is a *live* object of type t at address x. After the call $D_t(x, \ldots)$, we say there is a *zombie* object of type t at x, until another live object occupies at least one byte of memory in the address range $[x, x + \texttt{sizeof}(t))$.

Let p the value of be a pointer of type $s*$. If there is a live object of type t at some address x with $p - x \in \textsc{Offset}(t, s)$, then we call p *type safe*. If there is a live or zombie object with the same conditions, we call p *type consistent*. For convenience in later arguments, we also consider p to be type safe and type consistent if it is the Null pointer or if it points to zeroed memory of size `sizeof`(s). Note that type safety implies type consistency and is a strictly stronger property.

An *object allocation* is a heap allocation of some type t, obtained via `new t`, where t is not an array type. The actual use of the `new` operator is often deferred to a wrapper function returning a smart pointer such as `std::make_unique`. The actual memory allocation may be separated from the constructor call via the placement-new language construct, so long as the allocation size is the same, such as in `new (malloc(sizeof(t))) t`. If such an allocation can occur in a program, we say that t is *object-allocated*. Note that the resulting allocation is called an *object allocation* and will have a constant and statically known size. We call all other heap allocations *complex*. Most notably, this includes all variable-size allocations such as arrays or compound objects with a custom, dynamic memory layout.

Note that object allocations can contain arrays or data buffers as subobjects, although these have an inherent size limit.

2.3. Exploit mitigations

An *exploit mitigation* is a hardening measure provided by a compiler, library or runtime environment designed to make exploitation of memory safety issues harder. As such, it addresses security vulnerabilities in the time frame between their introduction and their eventual discovery and fix. Most mitigations address a specific class of issues which are known to have occurred commonly in the past. Important examples of widely deployed generic exploit mitigations include:

Data Execution Prevention / No eXecute (DEP/NX), a hardware-assisted mechanism to ensure that control flow never enters pages not marked as executable using a designated hardware bit. Applications can ensure via operating system APIs that none of their data pages have this bit set. This prevents attacks that redirect control flow directly to crafted machine code placed on the stack or heap.

Address Space Layout Randomization (ASLR), the randomization of the address space with the aim to force exploits to either learn the address space or work without forging full pointers. This is a secret-based mitigation and is rendered ineffective by exploits that leak information about the address space, or exhaust the entropy enough to place data at a controlled location – for example by causing a number of large heap allocations, a technique referred to as *heap spray*.

Stack cookies, a mitigation specifically designed to protect return addresses on the stack from stack-based buffer overflows, by placing a secret canary value before the return address, which is checked before returning.

Control Flow Integrity (CFI), an umbrella term for a variety of techniques that aim to ensure that the target of indirect control transfer instructions is within the set of targets that can occur during a valid program execution [3]. Different implementations consider different approximations of the precise target set. We categorize indirect control transfers into *forward edges* of the control flow graph, manifested in machine code by indirect jump/calls, and *backward edges* of the call graph, represented by return instructions. Notable forward edge software CFI implementations include the Microsoft Control Flow Guard (CFG) and Clang's `-fsanitize=cfi` instrumentation. Hardware support for CFI variants has recently been added to the AArch64 architecture with the introduction of ARM pointer authentication (PAC) and has be announced for Intel processors as Control-flow Enforcement Technology (CET), which is to appear in hardware in 2019 or 2020. CFI addresses exploitation techniques rather than the root cause of vulnerabilities: It thwarts exploits that are based on corrupting function pointers on the stack or heap to redirect the control flow of the target program.

Memory tagging, an extension of the classic flat memory model [4]. Every run of N bytes carries a short integer tag. A prominent example is the implementation to be released with ARM v8.5, which according to various sources will support 16-byte runs and 4-bit tags [5]. Pointers can optionally carry a tag in the unused high bits, in which case memory dereferences can be checked for matching tags. With hardware, compiler and standard library support, powerful and efficient mitigations for both spatial and temporal memory issues can be designed. One simple scheme is to tag every allocation with a random tag, possibly enforcing distinct tags for adjacent allocations, and to reserve one specific tag value for freed memory. Note that in the hardware-accelerated case, at least the CPU but possibly also the memory controller need to support this paradigm. It is unclear how efficiently it can be implement in software on commodity CPUs without hardware support.

Type-consistent memory reuse, a concept that was first introduced by [6] and implemented more efficiently by [7]. The two implementations differ significantly, however share the common insight that C programs developed according to modern software engineering practices mostly deal with well-typed, structured data – a property clearly also shared with modern C++ programs that have a much more expressive type system at their disposal. [6, 7] propose type-aware allocators which ensure that no two allocation requests for different types ever overlap. This breaks exploitation techniques for temporal safety issues which are based on reclaiming freed memory with objects of different types. While the fully generic solutions have not found widespread adoption, simple variants based on placing certain groups of objects on separate heaps are deployed in some security-critical software products such as the Windows kernel (GDI object isolation), the Chromium browser (PartitionAlloc) and WebKit (Gigacage, IsoHeaps).

Specific mitigations are often employed in large code bases to make known classes

of bugs outright unexploitable, or prevent known exploitation techniques. Sometimes, special-purpose mitigations are implemented to work around issues outside of the control of software vendors, such as timing side channels in CPUs.

2.3.1. Generic exploitation of temporal memory safety issues

To give an intuition on why type-consistent memory reuse is effective as a mitigation, consider a classic use-after-free vulnerability and a potential exploit:

1. Cause "victim" object to be freed while a dangling pointer to it still exists. This is often informally referred to as *triggering* the bug.
2. Allocate a lot of objects of the same size as the victim, but a different type, hoping that one of them will reclaim the freed space and replace the victim.
3. Cause a type-unsafe reuse of the dangling pointer.

This yields a powerful type confusion situation, because the attacker can chose the object type to confuse the victim with. The replacement object could even be fully controlled, as is the case with string data or binary blobs. The further steps highly depend on the victim object and the target program. Type-consistent memory reuse prevents this generic exploitation technique by breaking step 2 above.

2.4. Hypothesis

We make a number of conjectures which we will carefully investigate in the following:

1. Type-consistent memory reuse and more generally, aggressive separation of heap regions with different types of objects is a low-cost approach to address temporal memory safety bugs. Even if dangling pointers are not detected or prevented, generic exploit techniques are rendered ineffective. If pointer alignment is enforced, exploiting temporal vulnerabilities often becomes impossible.

2. Modern exploits rely heavily on violating the type safety of heap pointers. Hence, enforcing type safety on heap pointers makes exploiting certain vulnerabilities considerably harder and sometimes impossible. Even partial type safety, for example only for a subset of pointer types, is useful to thwart a large number of potential exploits, if that subset includes types that are essential for exploitation, or inherently dictated by the used vulnerability.

3. Grouping constant-size object allocations by type leads to a regular type structure in each of the separate regions. Type-related meta-data can be compressed easily because it repeats modulo the object size.

2.5. Related work

While most of the recent memory safety mitigation research is focused on control-flow integrity, there exists prior work on the mitigation of specific classes of vulnerabilities, including type confusion vulnerabilities in C++ programs. Examples of this include CaVer [8] and TypeSan [9]. They provide a way to check object types at runtime, using separately stored type information. Both mitigations add runtime type checks to all explicit static and dynamic casts in the program. However, they do not address the case of type confusions resulting from deliberately corrupted pointers on the heap and can not defend against those.

The EffectiveSan sanitizer [10] addresses a superset of the vulnerabilities that we are interested in mitigating: Its instrumentation is able to detect the kinds of pointer misuses and bad casts that are essential for exploitation. However, it stores the necessary meta-information in-band, alongside objects on the heap, and can thus not be trusted in the presence of spatial or temporal memory corruption. Furthermore, EffectiveSan is designed to detect certain kinds of memory safety errors such as out-of-bounds accesses as soon as they occur, hence requiring much more complete instrumentation of the program. For our purposes it is enough to instrument locations where a corrupted pointer is about to be used.

Both TypeSan and EffectiveSan are more generic than our proposed scheme due to the fact that they support stack and global memory as well as heap memory. We focus on heap memory safety and optimize for the common case where few pointers to the stack or to writable global memory are stored on the heap.

CFIXX [11] is a recently proposed approach which addresses the same core problem we are trying to address, but in a less generic fashion: They introduce the concept of Object Type Integrity, which in essence is a compiler mitigation that instruments all virtual call sites to add dynamic type checks. This is done specifically to counter COOP-like attacks, but does not address the more general issue of data-only attacks and CFI bypasses based on arbitrary read/write primitives (which may be obtainable without virtual method calls).

Mid-fat pointers [12] provide a way to encode the location of metadata such as type information inside pointers itself. It is unclear how this approach could possibly deal with sub-objects or inheritance, where the address of a sub-object is taken, and thus if it is even applicable to C++ programs. Additionally, this approach restricts the effective address space of the target program to 32 bits. In their related work section, the authors outline the state of the art in dynamic pointer verification and spatial memory safety quite exhaustively, including their older METAlloc metadata management scheme [13].

3. Design and Implementation

In this section we introduce our attacker model and security policies that defend against it. We proceed to describe a data structure necessary to enforce these policies. Applying them to an existing code base requires analyzing and instrumenting the program, which we solve by means of a plugin for the Clang compiler.

3.1. Attacker model and mitigation goal

Let D be a set of *dangerous types* chosen arbitrarily, for example based on empirical observations about past exploits. Certain practical considerations restrict the choice of this set, the details of which will be explained later. We consider an attacker with the ability to forge or corrupt a *dangerous pointer* of type $t*$ on the heap, where $t \in D$. Common examples of software bugs that can lead to this ability include: treating uninitialized memory as a pointer; fetching a pointer from an already freed memory location via a dangling reference; fetching a pointer from outside a data structure via an out-of-bounds read, or corrupting a pointer on the heap via an out-of-bounds write.

Exploitation of this primitive is often possible by forging an object of type t inside fully attacker-controlled data, or replacing the pointer with a potentially misaligned pointer to an object of different type. This can in turn lead to more powerful primitives such as writes to arbitrary addresses or direct control flow hijacks via function or vtable pointers. Our goal is to ensure dangerous pointers are type safe or at least type consistent.

3.2. Security policies

We introduce the concept of a *safe heap region* first, which is one that is separated from other heap regions by unmapped memory, ensures type-consistent memory reuse, and where no complex allocations are served from. We assume that safe heap regions require less instrumentation than the rest of the heap, thus saving instrumentation overhead. Some empirical observations can be made about safe heaps:

- Due to type-consistent memory reuse, pointers in the safe heap are generally type consistent, unless spatial memory corruption occurs;
- Because there are only constant-size data structures allocated in a safe heap, it is unlikely that data structures in the heap itself are affected by spatial

memory safety issues – we will discuss this aspect in more detail as part of our evaluation in subsection 4.3.2;

- An attacker would thus have to perform a near-arbitrary write to control a dangerous pointer inside a safe heap, since the write would have to "skip" over a region of unmapped memory between the safe heap and other mapped memory regions.

A heap region that is not safe is considered *wild*. Examples of such heap regions are such regions where objects of varying size are allocated, for example in a first or best fit fashion, or where objects of the same size but different types are allocated. Intuitively, type consistency of pointers into such a heap can easily be violated by reclaiming freed memory, or by an out-of-bounds access to variable-sized objects such as arrays. We propose the following security policies of decreasing strictness:

- *Pointer type safety/consistency* dictates that all dangerous pointers that are fetched from the heap are checked for type safety/consistency before they are dereferenced.
- *Wild pointer type safety/consistency* dictates that all dangerous pointers that are fetched from wild heap regions are checked for type safety/consistency before they are dereferenced. Pointers fetched from safe heap regions are not considered dangerous.

Note that an implementation of wild or non-wild pointer type consistency can be extended to wild or non-wild pointer type safety easily by zeroing out every object after it has been freed. We will evaluate the performance overhead of this operation in subsection 4.2.3.

3.3. Scope

We want to implement the security policies defined above in an existing C++ program. Due to the complex semantics of C++, we do not expect this to be possible in a fully generic way while still maintaining a reasonably low runtime performance overhead within the scope of this work alone. Our goal is to develop a semi-automated process which can be applied as a prototype to an existing code base that is too large to manually audit or instrument as well as heavily optimized for performance: This is precisely the type of code base most likely to contain memory corruption vulnerabilities.

This restricts our design space to solutions based on very mature tooling based on state-of-the-art C++ compilers. We chose to use the Clang compiler[1] as a basic building block due to its perceived higher flexibility and extensibility when compared with the GNU Compiler Collection[2].

[1] `https://clang.llvm.org/`
[2] `https://gcc.gnu.org/`

In the scope of this work, we will restrict ourselves to transformations and simple analysis based on the Clang AST, which is a rather high-level construct, rather than on a lower-level intermediate representation such as LLVM IR. While this limits our options in terms of static analysis, it will allow us to describe and successfully develop a working prototype for a large web browser library, and obtain data on practical considerations like instrumentation completeness and runtime performance overhead.

3.4. Type consistency data structure

Let T be the set of C++ types in a target program. We present a process that can be applied to a codebase in order to support type consistency checks. The general steps are as follows:

1. Accept a set D of types as input which are deemed dangerous by the user;
2. Determine the subset $\textsc{Checked} \subseteq D$ of maximum size for which type consistency can be checked efficiently using the proposed data structure;
3. Add library code to the target program to support the program instrumentation;
4. Compile the target program under a transformation pass which applies the instrumentation.

The resulting program has access to a data structure that can efficiently answer type consistency queries: Given a pointer p of type $s*$ with $s \in \textsc{Checked}$, it can determine whether p is type consistent.

The fundamental primitive by which this is achieved is *type isolation*: For each $s \in \textsc{Checked}$ and for each of its object-allocated super-records $t \in \textsc{Super}(s)$, object allocations of type t are served from an *isolated heap* which is initially zero-initialized and where only objects of type t are allocated, at regular offsets $0,\ \texttt{sizeof}(t),\ 2 \cdot \texttt{sizeof}(t)$ and so forth. The set of types placed in isolated heaps is called Iso. This leads to a repetitive type structure within each heap: Given a pointer p into an isolated heap for some type t, it can be decomposed into three parts $p = H_t + k \cdot \texttt{sizeof}(t) + \Delta$, where H_t is the start address of the isolated heap, k is the index of the object which p points into, and $0 \leq \Delta < \texttt{sizeof}(t)$ is the inner-object offset to which p points. If p has type $s*$, and $\Delta \in \textsc{Offset}(t, s)$, then it is type consistent. Note that isolated heaps are safe heaps according to our definition from above, as long as they are allocated with unmapped memory surrounding them.

With access to a data structure that represents Offset, the program can now check type consistency for pointers to Checked types. Note that type isolation is an example of type-consistent memory reuse which was proposed by [6] and [7] and is already implemented in various software products. Type-consistent memory reuse has the property that type consistency of pointers is preserved over time, and dangling pointers remain type consistent.

3.4.1. Algorithm

Given a C++ program and the set of types D, we will now describe how to determine CHECKED $\subseteq D$ and the set of object-allocated types ISO which have to be placed in isolated heaps in order for the proposed data structure to work. A number of circumstances restricts our choice of these two sets, as described in the following.

Let BROWN $\subseteq$ BLACK $\subseteq$ ALLOCATED and RED be subsets of T defined as follows: $t \in$ ALLOCATED if t is object-allocated. $t \in$ BROWN if t is object-allocated, but cannot be put into an isolated heap due to implementation-specific restrictions. Our suggested implementation, as described in subsection 3.5.1, entails the following restrictions:

- t is not a type defined via `class` or `struct`;
- t is a *local class*, defined within a C++ function;
- t is a *template class*;
- `sizeof`(t) is too large;
- t has derived classes but no virtual destructor;
- There exists a helper function that allocates t manually and overloads `operator delete` to be compatible with that allocation mechanism;

$t \in$ BLACK if t is object-allocated, but cannot be put into a *checked* isolated heap, because $t \in$ BROWN or because other implementation-specific restrictions prevent us from building the OFFSET data structure on t (see subsection 3.6.1).

$t \in$ RED if pointers to t cannot be checked for type consistency. This occurs in one of the following cases:

- t is contained in a complex allocation;
- $t \in$ BLACK;
- $s \in$ RED for some super-record s of t.

The last case implies transitivity of RED with regard to type composition, i.e. if some $t \in$ RED then all sub-records of t are as well. These definitions let us formulate the restrictions imposed on CHECKED and ISO:

- CHECKED $\cap$ RED $= \emptyset$
- SUPER(CHECKED) $\cap$ ALLOCATED $\subseteq$ ISO

The maximal choice of CHECKED is thus CHECKED $= D \setminus$ RED. We are free to choose isolated types in ALLOCATED outside of SUPER(CHECKED), in order to ensure type-consistent memory reuse in the corresponding heap region. Particularly interesting candidates are those types in $D \setminus$ BROWN.

3.5. Program analysis and instrumentation

In order to understand the types defined and used by a given target program and to identify dangerous pointers, we require compiler-specific information. We developed a Clang extension that hooks into a successful compilation process of the program and extracts all relevant information from the final typed C++ AST, which includes:

- The memory layouts assigned to all **`class`** and **`struct`** definitions by the compiler;
- Information about pointer members of classes and structures;
- Information about object allocation sites.

From this we can compute the Offset relation and the Checked and Iso sets as described above. We also compute two additional sets for optimization purposes: Wild contains all types that can occur inside a wild heap region. This set is computed by considering all uninstrumented allocation sites and the object types which those allocations can contain. WildPointers contains all types t for which a pointer $t*$ can occur inside a wild heap region.

We assign a unique identifier $\mathrm{ID}_t \in \mathbb{N}$ to every relevant type $t \in T$. All of this information is stored in a simple text-based database. A second compilation pass of the target program is performed which applies transformations to the AST in order to add the necessary type isolation and pointer check instrumentation. The instrumentation makes use of a small runtime and template library, as well as a special-purpose allocator library to implement the isolated heaps. In the following we will give a short overview over the AST changes performed to the target code base.

3.5.1. Per-type metadata and allocators

The first transformation that the instrumenting compilation pass applies works on a **`class`**/**`struct`** definition level, illustrated by example Figure 3.1. The static member variable `t::type_id` corresponds to the value ID_t defined above. In the case where the **`new`**/**`delete`** operators for t are already overloaded, the existing overloads are replaced. The decision to perform this instrumentation on an AST level induces a number of restrictions: since the resulting AST must correspond to a legal C++ program, language semantics dictate that

- t must be a compound type defined via **`class`** or **`struct`**;
- t cannot be a local type, defined inside a function, since those cannot have static members;
- t cannot be a template class or partial template specialization. It can however be an explicit full template specialization. The problem with underspecified (i.e. template argument-dependent) template definitions is that there is no

```
class t {
  int x;
  // ...
};
```

→

```
class t {
public:
  static constexpr size_t type_id = 1337;
  static IsoHeap<A> isoheap;
  void* operator new(size_t) {
    return isoheap.allocate();
  }
  void operator delete(void* o) {
    return isoheap.deallocate(o);
  }
  // ... more operator overloads
private:
  int x;
  // ...
};
```

Figure 3.1.: This example illustrates the transformations applied to relevant type definitions for type isolation and per-type metadata.

obvious way to specify a compile-time value for the `type_id` member, which is a function of the final concrete type. To the best of our knowledge, this would require applying AST transformations on a per-template-instantiation basis, which is non-trivial given the Clang programming interface. There is also no general way to forward declare t such that a type-level function can be used instead of a static member. To keep our prototype somewhat simple, we decided not to implement a more advanced solution.

- There can be no places where an object of type t is allocated in a custom way, expecting the object to be deleted later using a specific **`delete`** operator. As an example, assume the standard `free` function is used as the global **`operator delete`**. A constructor function could allocate space for an object via `malloc` instead of **`operator new`**, which will cause the program to break if **`operator delete`** is overwritten to use anything other than `free`.
- If t does not have a virtual destructor, it cannot have object-allocated derived classes, because the defined **`delete`** operator is invalid for pointers upcast from a derived class to $t*$.

The `IsoHeap` implementation might have additional restrictions such as a maximum object size. These restrictions define the set BROWN.

3.5.2. Instrumenting pointer fetches

Our goal is to implement the (wild) pointer type consistency policy, and as such we have to instrument all locations where pointers are fetched from the heap. In

```
struct t;
struct A {
  t* x;
  unique_ptr<t> y;
  void test() {
    this->x; // member access
    *y;      // smart pointer dereference
  }
};
vector<t*> vec(10);
t** ary = new t*[10];
A a;
void test() {
  vec[0];          // container access
  *ary;            // pointer to pointer
  a = *new A{};    // copy
}
```

Figure 3.2.: Examples of pointer fetches which must be instrumented by our second compilation pass.

C++ semantics this corresponds to dereferences of the form $*p$ where p has type $t**$ for some $t \in$ CHECKED, and p points into the heap. An overview of multiple different manifestations of this pattern are illustrated by Figure 3.2. The last line highlights a notable special case: If a pointer is copied from the heap to another memory region, it must be checked during this operation according to our security policy. We solve this approximately by using the C++ type system and applying the transformations illustrated in Figure 3.3. The example shows the introduction of a pointer-like wrapper class `CheckedPtr<t>` which replaces the usage of the raw pointer type `t*`. It represents a pointer which can only be read after checking it according to our security policy.

`unwrapChecked` and `refChecked` perform the task of checking the wrapped pointer value and returning the value as an rvalue and lvalue, respectively. The decision which wrapper function to use is based on whether the result is implicitly cast to an rvalue in the AST, indicating that the pointer value is only read from rather than written to. `refChecked` is unsafe by definition: Since it returns a reference to a checked pointer rather than the pointer itself, if this reference is used to later read the pointer value, there is window in time between the check and the use of the value during which the value could have changed. Storing the reference and retrieving the pointer value later, after it might have already changed, is unsafe. Hence our instrumentation only introduces this function as a last resort, and prefers to rely on operator overloads such as `CheckedPtr::operator=` and `CheckedPtr::operator+=` where possible. As a standalone smart pointer type, `CheckedPtr` behaves according to the usual pointer semantics. It has a copy constructor that performs the required type consistency check.

```
struct A {
  t* x;
  t* get() {
    return x;
  }
  void change() {
    x += 10;
  }
};
template <class T>
struct B {
  T x;
  T get() {
    return x;
  }
};
vector<t*> vec;
void test() {
  t* x = vec[0];
}
```

```
template <typename t>
t* unwrapChecked(
    const CheckedPtr<t>& x);
template <typename t>
t*& refChecked(CheckedPtr<t>& x);

struct A {
  CheckedPtr<t> x;
  t* get() {
    return unwrapChecked(x);
  }
  void change() {
    refChecked(x) += 10;
  }
};
template <class T>
struct B {
  MakeCheckedPtr_t<T> x;
  T get() {
    return refChecked(x);
  }
};
vector<CheckedPtr<t>> vec;
void test() {
  t* x = unwrapChecked(vec[0]);
}
```

Figure 3.3.: This examples illustrates the transformations applied to raw pointers to types in CHECKED. `MakeCheckedPtr_t` is a type-level function the declaration of which is left out for brevity. Its purpose and functionality is explained in subsection 3.5.2.

To instrument template class fields which can be pointers and non-pointers in different instantiations, the `MakeCheckedPtr_t<t>` type-level function is used which resolves to `CheckedPtr<remove_pointer_t<t>>` if `t` is a checked pointer type, or just `t` otherwise. Both `unwrapChecked` and `refChecked` fall back to the identity function in case the argument is not a `CheckedPtr`, so both can be used with an argument of type `MakeCheckedPtr_t<t>`.

This conceptually simple approach allows for violations of the security policy in certain scenarios. An example of this would be a function of type `f(t*& x)` with `t` ∈ CHECKED and an instrumented call site `f(refChecked(x))`. Here, `x` is checked at the call site. `f` however receives a reference to a potentially dangerous pointer and there is no guarantee that it is not used at a later time without a check. The obvious solution to this problem would be to turn every instance of `t*` into a `CheckedPtr<t>` – in the given example it would mean to change `f`'s parameter type to `CheckedPtr<t>&`. This is a highly invasive transformation and would likely require additional compilation passes, as well as introduce a lot of unnecessary pointer checks. We believe in order to employ this approach, at least comprehensive control and data flow analysis would be required to identify pointers that provably cannot originate from the heap, which is outside of the scope of this work.

For our prototype implementation, we used a different, less generic approach: Where possible, we avoid the `refChecked` wrapper function completely, and instead overload the assignment operators of the `CheckedPtr` smart pointer type. This means that the problematic pattern where a check is performed but a reference is passed on is not introduced. Instead, compilation errors are produced at the affected locations, which need to be investigated manually as a result.

3.6. Pointer checking algorithm

Start address	End address	Heap contents
`0x2000,0000,0000`	`0x3000,0000,0000`	Unsorted isolated heaps
`0x3000,0000,0000` $+ \mathrm{ID}_t \cdot$ `0x800,0000`	`0x3000,0000,0000` $+ (\mathrm{ID}_t + 1) \cdot$ `0x800,0000`	Fixed isolated heap for type t
`0x3800,0000,0000`	`0x4000,0000,0000`	Everything else (wild heap)

Figure 3.4.: Address space layout of the heap. For each $t \in$ ISO with $\mathrm{ID}_t < 2^{16}$, a 128 MiB heap region is pre-allocated. The entire heap spans the address space `0x2000,0000,0000`–`0x4000,0000,0000`, which amounts to a total of 32 TiB. This address interval does not collide with any default OS mappings on Windows, macOS or Linux. In between the regions, a configurable amount of pages is protected by setting the page permissions to zero.

Input: p, a value of type $t**$ with $t \in$ CHECKED.
Output: The pointer value $x = *p$ if it is type consistent, or else a program crash.
if $t \notin$ WILDPOINTERS and only wild pointer consistency required,
then return x (this situation is known at compile time)

1: **if** p not on heap or $x =$ NULL **then**
2: // No need to check according to policy
3: **return** x
4: **end if**
5: **if** only wild pointer consistency required **and** and p not on wild heap **then**
6: // No need to check according to policy
7: **return** x
8: **end if**
9: $\text{start}_t :=$ `0x3000,0000,0000` $+ \text{ID}_t \cdot$ `0x800,0000` (known at compile time)
10: **if** $x \in [\text{start}_t,\ \text{start}_t +$ `0x800,0000`$)$ **then**
11: // Fast path, just ensure alignment
12: **assert** $x - \text{start}_t$ **mod** sizeof$(t) = 0$
13: **return** x
14: **end if**
15: **if** x points inside pre-allocated isoheap region of another type **then**
16: // Compute type of containing object
17: $\text{ID}_u := (x -$ `0x3000,0000,0000`$)$ **div** `0x800,0000`
18: $\text{start}_u :=$ `0x3000,0000,0000` $+ \text{ID}_u \cdot$ `0x800,0000`
19: // Decompose $x = \text{start}_u + k \cdot \text{sizeof}(u) + \Delta$
20: $\Delta := (x - \text{start}_u)$ **mod** sizeof(u)
21: **assert** $\Delta \in$ OFFSET(u, t)
22: **return** x
23: **end if**
24: // Invalid pointer detected
25: Crash the program

Figure 3.5.: Algorithm `checkAndFetch` for (wild) pointer type consistency check.

To provide an efficient method for the pointer-checking code to distinguish between different heap regions, we lay out the address space as depicted in Figure 3.4. Checking whether an address is on the heap at all, or within any of the specific heap regions, amounts to a simple bit-masking operation and comparison. Figure 3.5 shows the algorithm used for checking and fetching a pointer from the heap, given the address where it is located.

3.6.1. Type metadata data structure

For pairs of types $t \in \textsc{Iso} \cap \textsc{Super}(\textsc{Checked})$ and $s \in \textsc{Checked}$, we need a data structure representing $\textsc{Offset}(t, s)$. Let n be the number of types covered by our analysis pass. The data structure we chose is a matrix M of dimension $n \times n$ where each element is a bitmap of size m. For $a := \mathrm{ID}_s$ and $b := \mathrm{ID}_t$, the i-th bit in $M_{a,b}$ is set to 1 if and only if $8 \cdot i \in \textsc{Offset}(t, s)$. As such, $M_{a,b}$ compactly represents the set $\textsc{Offset}(t, s)$. Due to the constant size of each bitmap, we can place each matrix entry at a memory location solely dependent on the values of a and b. Thus, given the type identifiers of s and t as well as an offset Δ, only a single memory access is required to establish whether $\Delta \in \textsc{Offset}(t, s)$, which is precisely the information required by our pointer check algorithm introduced above.

This data structure has two inherent restrictions: It can only handle types $t \in \textsc{Iso}$ of size at most $m \cdot 8$, and it can only handle sub-objects whose offset from the beginning of the containing allocation is aligned to 8 bytes. In our implementation we choose $m = 1024$, which covers objects of up to size 8192 bytes. Object types above this size, or those with checked sub-objects aligned to 4 bytes or less fall into the Black set of types.

The memory footprint of this data structure is reasonable due to the fact that almost all entries of the matrix are zero and can be mapped in memory without reserving physical pages. We will measure the precise resident set in our evaluation section subsection 4.2.4.

3.7. Case Study: Instrumenting the WebCore library

To evaluate the viability of our proposed data structure and compiler instrumentation, we were considering open-source projects with the following properties:

- Large and modern enough to contain many examples of rare uses of C++ language features and standard library – we want to make sure that we can handle all kinds of special cases;
- Supports compilation with current Clang, under the newest C++ standard (C++17);
- Highly security-critical, ideally with remote attack vectors;
- Actively developed and audited for security issues;

- Recent data about public vulnerabilities and well-established vulnerability triaging processes.

The most obvious choices are the open-source web engines Blink, Gecko and WebKit that are part of the major web browsers by Google, Mozilla and Apple, respectively. Out of those, Blink and WebKit already have implementations of isolated heaps, and WebKit has an opt-in per-class isolation mechanism called *IsoHeap* which is ideal as starting point for our prototype. At the time of writing, the WebKit code base contains approximately 3.3 million lines of C++ code, including 1 million lines inside header files.

What makes WebKit particulary appealing as a case study is the fact that in 2017 and 2018, more than 40 security bugs were reported in WebKit by Ivan Fratric and Jung-hoon 'lokihardt' Lee, who run browser fuzzing projects at Google, and other examples of security-critical bugs are publicly documented. All of these issues are caused by memory corruption. This gives us insights into the nature of common security vulnerabilities in this code base. We have to be careful to interpolate from one code base to others however, as well as from one security researcher to others, but this is currently the best we can do with public information.

The list of recent public security bugs in WebKit can be grouped into three categories: Bugs inside the JavaScript engine (JavaScriptCore, JSC), bugs inside the main HTML/DOM engine (WebCore), and bugs in third-party libraries such as libwebrtc. JSC vulnerabilities tend to be caused by logic flaws that are exploited very differently on a case-by-case basis. Objects in JSC are managed by a garbage collector, store dynamic type information and often contain a dynamic amount of inline data. As such their characteristics are very different from the traditional, fixed-size C++ objects that we are focusing on in this work. We believe this component deserves and requires its own special-purpose mitigations. For our case study, we focus on the WebCore library instead: The set D of dangerous types is chosen as all the types in the `WebCore` namespace.

3.7.1. Isolated heap allocator

For our prototype, we used the *bmalloc* allocator implementation included in the *WebKit* library. It allocates isolated heap memory in pages of size 16 KiB. Chunks are allocated right next to each other from bottom to top within a page, using a simple bump allocator. A bitmap located at the beginning of each page keeps track of chunk liveness. Once the most recently allocated page fills up completely, the bitmap is used to build up a singly linked free list of unused slots. The forward linked list pointers are stored inside the free chunks, XOR-ed with a random value, which is a slight violation of our definition of type safety and consistency, which would require a zero instead of a random value in this location.

3.7.2. WebKit containers and allocators

WebKit bundles its own support library called WTF – presumably an abbreviation for *WebKit Template Framework*. It provides various container types, the most common of which are `Vector`, `RefPtr`, `Ref`, `HashMap` and `HashSet`. `RefPtr` and `Ref` implement reference-counted NULL-able and non-NULL-able smart pointers, respectively, which are used instead of `std::shared_ptr`.

WebKit uses the allocator provided by the C++ standard library in some places, but also bundles its own allocator bmalloc. bmalloc is used to implement the FastMalloc, Gigacage and IsoHeap mechanisms: FastMalloc supports simple `malloc`-style allocations of arbitrary size. Chunks of similar size are placed in the same heap pages. The `WTF_MAKE_FAST_ALLOCATED` macro can be used in a class definition to generate **`new`**/**`delete`** operator overloads backed by FastMalloc.

Gigacage groups together allocations of certain types in a large heap regions of size more than 2^{32} bytes. At the time of writing, all JavaScript array buffers are allocated in one such region. Gigacage was most likely introduced as a mitigation against timing side channel attacks: array accesses with 32-bit indexes can not point outside of the corresponding heap regions and leak data from there by design. Currently it is not used by WebCore. IsoHeap is an implementation of a type-isolated heap which is described in subsection 3.7.1. It is enabled for a specific class or structure type using the `WTF_MAKE_ISO_ALLOCATED` macro. At the time of writing, 362 types in the `WebCore` namespace carry this annotation.

3.7.3. Instrumentation and library code

We implemented the instrumentation described in section 3.5 with the WebKit codebase in mind and added the necessary library support code to the WTF library. During development of the Clang compiler plugin we encountered a bug in the source location functionality of the Clang AST which prevented us from applying the `MakeCheckedPtr_t` transformation to some very generic template classes such as `RefPtr` and `std::unique_ptr`. We worked around these issues by instrumenting these classes using manual source code changes. Also, we manually added instrumentation to `Vector<t*>` where $t \in$ CHECKED, so that it behaves like `Vector<CheckedPtr<t>>` in client code, but without requiring checks whenever copying a pointer from one heap location to another, for example during resize operations. This is valid due to the fact that the destination of such copies is always a wild heap location, and the pointer is not dereferenced in the process.

Other library changes include a drop-in replacement for the memory manager `VMAllocate`, to implement the address space layout depicted by section 3.6, as well as the addition of the pointer checking implementation itself. In some places we manually fixed compiler errors that resulted from our second compiler pass. For example, due to the way we instrument `Vector<t*>`, loops of the form **`for (auto*`** `it : <vector>)` had to be changed to **`for (auto`** `it : <vector>)` with the type of `it` changing from `t*` to `CheckedPtr<t>`.

As a simple optimization if only wild pointer type consistency is required, we can use the information represented by the set WILD (see section 3.5) to avoid some instrumentation: Pointer fields contained in class or structure definitions which are not contained in this set require no instrumentation, since such a pointer can never occur on the heap.

3.8. Optimizations

In order to achieve a reasonable performance of our instrumentation, we put some optimization effort into the pointer check algorithm due to its omnipresence in the program. In particular, we replaced the costly division operation to compute inner-object offsets by cheaper multiplication operations followed by a right shift, a transformation also commonly employed by mainstream compilers to optimize division by constants.

Furthermore, for some types of pointers we implemented special cases specifically optimized for WebCore to avoid the usage of our generic `CheckedPtr` type. This includes `WTF::RefPtr`, `std::unique_ptr` as well as `WTF::Vector<T*>`. A good example of how special cases can lead to fewer checks is a `WTF::Vector` containing pointers: A strict implementation of pointer type consistency requires checking all pointers whenever the backing buffer of the vector is resized and all elements are copied to the new allocation. By special casing this operation, we can avoid the checks since it is clear that the pointers are not dereferenced, and they are copied from one wild heap location to another one, so they will be checked before their next use.

3.9. Limitations of our implementation

There are two major limitations of our current prototype implementation that we are aware of: One is the absence of pointer checks for raw references – i.e. `T&` and `const T&` in C++, where $T \in$ CHECKED, due to technical reasons related to the way we process the Clang AST. The second limitation is explicit black listing of some classes from being included in the CHECKED set, due to what we believe is a bug in Clang when fetching the source location corresponding to certain expressions. Specifically, in the Clang AST we are looking for `clang::MemberExpr` nodes that access pointer fields of a `class` or `struct`. However, for some template classes it seems that Clang returns the wrong source location for these expressions, preventing the rewriting required by our instrumentation. We worked around this issue by detecting the affected instances with a heuristic and excluding the corresponding types from the CHECKED set. Unfortunately this affects some types high up in the type hierarchy such as `WebCore::Node`. Reducing this problem to a manageable subset of the source code – for purposes of root causing the issue and reporting it to the LLVM maintainers – has so far eluded us, since it appears to occur mostly

within template classes that are instantiated with multiple very commonly used types. However, our security evaluation in subsection 4.3.1 shows that both of these limitations do not seem to effect any of the real-world vulnerabilities we analyzed.

4. Evaluation

In the previous chapter, we described a prototype implementation of our proposed pointer type safety and consistency policies. To reiterate, this entailed changes to the target program, applied at compile time: As a fundamental building block, we put many extra types into isolated heaps according to the result of an algorithm run on the C++ type structure of the program. Besides increasing type-safe memory reuse in the program in and by itself, this has the added advantage of allowing us to perform efficient type checks within these regions. A second instrumentation is performed at compile time to add such pointer type checks.

In the following we will evaluate the result of our WebCore instrumentation for wild pointer type safety and consistency with a focus on performance, memory efficiency and effectiveness as a security mitigation. To evaluate the latter, we will analyze recently patched security vulnerabilities in WebCore and compare exploitation options with and without our instrumentation. We will also analyze attacker models under which our approach breaks fundamentally, and propose theoretical solutions to some of these problems. All experiments were performed using our instrumented WebKit codebase, which is based on commit `736db5b1f5` from July 17th, 2018 in the Git mirror repository.[1].

4.1. Static analysis results

To interpret the performance data, we will first lay out basic information about the extent of the instrumentation. Our static analysis pass identified about 2.8 million relevant C++ types across the entire WebKit codebase. Amongst all types, we identified 12511 types that are object-allocated, and 16791 for which there exists some other type that points to it. There are 1456 types which cannot be put in checked isolated heaps using our implementation for various reasons – this corresponds to the set Black described in subsection 3.4.1. Furthermore, the size of Red, the set of types which cannot be checked – the reasons for which are also laid out in subsection 3.4.1 – is 5520. As a result, we identified 2089 types which we can automatically place in isolated heaps, constituting the set Iso. Out of these, 1994 have full type information available for pointer checks. To put this into perspective, the original WebKit codebase only uses isolated heaps for 362 or 17.3% of these by default.

The set of dangerous types D was chosen to include all types in the `WebCore` namespace, which amounts to 1465 types. The type information available for the

[1] `https://github.com/WebKit/webkit/commit/736db5b1f5`

majority of the isolated heaps allows us to check pointers for |CHECKED| = 1190 types. This includes many types which have been subject to memory corruption bugs, or which have been used as part of memory corruption exploits against WebCore in the past, as we lay out in subsection 4.3.1. For 439 out of the 1190 checked types t, pointers of type $t*$ can only occur inside safe heap regions, which allows us to optimize pointer checks away if only wild pointer safety/consistency is desired.

The total size of our offset data structure amounts to 222 KB, however since it is initialized lazily and due to its non-local structure the actual virtual memory footprint differs in practice. We measured it dynamically in our different performance benchmarks.

4.2. Dynamic evaluation

In this section we describe the dynamic experiments performed using the WebKit-GTK+ Minibrowser, the open-source browser which is shipped as part of the WebKit codebase. By running our instrumented build on several benchmarks, we collected dynamic data about the pointer instrumentation such as the number of checked pointer fetches, categorized by the different types of pointers, and by the different paths taken in the pointer fetch algorithm. By comparing the runtime performance, memory usage and code size against an uninstrumented build, we gathered insights into the overhead of the two essential parts of our instrumentation, which are the automatic type isolation and the pointer type consistency checks.

All experiments were performed using a builds which implement one of the wild pointer safety or consistency policies section 3.2, which we think provide the best trade-off between performance and security.

4.2.1. Experimental setup

We selected three different benchmarks that reflect modern usage of the DOM and other WebCore features:

- The `gmail` experiment is performed as follows: We log into our own Google Mail account, then open a new tab and browse to the inbox page.[2] We measure the time from pressing ENTER until the load indicator in the tab bar stops spinning, i.e. the time to load the page fully. Lower page load times indicate better performance.

- The `dromaeo` experiment consists of a full pass through the DOM-related benchmarks in the Dromaeo browser benchmark suite.[3] The benchmark reports the number of tests performed per second, hence higher numbers indicate better performance. Since some of the individual benchmarks cause a large amount of

[2] `https://mail.google.com/`
[3] `http://dromaeo.com/?dom|jslib|cssquery`

allocations which crash the Minibrowser due to the renderer process running out of memory, we removed the "DOM Events (jQuery)" and the "DOM Modification" tests for our experiments.

- The `speedometer` experiment consists of a full pass on the Speedometer 2.0 benchmark.[4] This benchmark is designed to capture the interactions of modern web frontend frameworks with the DOM and other web features and was recommended to us by a lead Apple engineer as a meaningful real-world benchmark for overall WebKit performance.[5] The benchmark runs 480 different experiments one after each other, and reports the number of experiments completed per second. Hence, higher numbers indicate better performance.

All benchmarks were conducted using three differently configured versions of the same WebKit codebase, based on commit `736db5b1f5` from July 17th, 2018 in the Git mirror repository.[6]. The builds were performed using our modified Clang version, built from HEAD on Ubuntu 18.04.1. The configurations are as follows:

- `vanilla` is a regular WebKitGTK+ release build.
- `instrument` is a WebKitGTK+ release builds with instrumentations to implement our wild type pointer consistency policy.
- `instrument-clobber` is identical except for additional code which clobbers all objects in isolated heaps using `memset` upon `free`. Thus it implements the stronger wild pointer type safety policy.
- `unchecked` is a WebKitGTK+ release build with our automatic type isolation instrumentation active, but without object clobbering and without the pointer fetch instrumentation. Specifically, the difference to `instrument` is the lack of pointer checking. We added this target to isolate the performance costs of type isolation against object clobbering and our pointer checking algorithm.
- `unchecked-clobber` is the same build but with the above-mentioned clobbering enabled.

We also used a debug build of the `instrument` target, extended by a mechanism to collect statistics about memory usage of the offset data structure, as well as detailed statistics about the pointer fetch algorithm. For obvious reasons it is not used in the performance benchmarks.

The experiments were conducted inside a Ubuntu 18.04.1 virtual machine running in VMware Workstation 15.0.2 on a Windows 10 host. The VM was configured to use all eight cores (and 16 hyper-threads) of the AMD Ryzen 7 1800X CPU available to the host, as well as 24 GB of DDR4 RAM. We performed each experiment at least 5 times on each target. In between runs we restarted the browser process to return to a clean process state.

[4] `https://browserbench.org/Speedometer2.0`
[5] `https://twitter.com/filpizlo/status/1049723484857806848`
[6] `https://github.com/WebKit/webkit/commit/736db5b1f5`

4.2.2. Instrumentation results

In the `speedometer` benchmark, 452 of the isolated types were allocated at least once by the program. The distribution of classes of pointers fetched via the pointer check algorithm is as follows: raw pointers (55.0%), `WTF::RefPtr` (34.4%), `std::unique_ptr` (10.3%) and `WTF::Vector<T*>` backing buffers (0.3%). In total, the pointer fetch algorithm `fetchAndCheck` is called 1.1 billion times. In 24.6% of cases, the pointer is identified to not be located on the heap and thus does not need to be checked. In another 67.4% of cases, the pointer is not located on a wild heap, and since we implement the wild pointer type safety policy, does not need to be checked either. These two situations occur when our static analysis cannot prove that the given pointer field is always located on the stack or non-wild heap, respectively. The remaining 8.0% of cases actually require checking the value of the requested pointer, and the different cases of the checking algorithm are distributed as follows: NULL pointer (6.1%), fast path (77.6%), slow path with $\Delta = 0$ (16.2%), as well as slow path with $\Delta \neq 0$ (0.1%).

In the `dromaeo` benchmark, only 358 isolated types are allocated during the entirety of the experiment, and `fetchAndCheck` is invoked 6.9 billion times. The distribution of checked pointer types is: raw pointers (96.9%), `WTF::RefPtr` (2.4%), and `std::unique_ptr` (0.6%). 98.9% of pointers are not located on the wild heap and hence are not checked, which is an even higher percentage than in the `speedometer` case. Of the remaining pointer checks, 8.7% return NULL, 77.3% are resolved via the fast path, and 13.2% are resolved via the slow path for $\Delta = 0$.

In the `gmail` benchmark, to the best of our knowledge, the precise numbers differ slightly from the `speedometer` benchmark, however the overall patterns are the same. Unfortunately our debug build of WebKit – which collects all of the above information – crashes in the middle of loading the Google Mail inbox page due to some unexpected timing gap introduced by the slow maintenance of the pointer logs; hence we were not able to collect the full trace of an entire page load.

One interesting observation here is that it appears as if better static analysis could help eliminate the large portion of calls to `fetchAndCheck` that can successfully return early without even considering the actual value of the pointer, for example because a pointer is not located on the heap.

4.2.3. Wall clock results

The detailed timing results for each combination of target and benchmark can be found in Table 4.1. In the `gmail` benchmark, we can observe a 3.7% timing increase induced by our automatic type isolation, while the pointer checks induce another 6.3% increase, yielding a combined 10.3% wall clock time increase of the `instrument` build compared to the `vanilla` build. The insights into the overhead of object clobbering on free are not conclusive given our data, since they range from 5.0% to 8.3% between the `instrumented` and `unchecked` build.

In the `dromaeo` benchmark, we measure throughput instead of time, so larger

numbers are better. We observed a 2.4% increase in throughput in the `unchecked` compared to `vanilla`, which can likely be attributed to better cache locality and the simpler allocator used for IsoHeaps. The pointer checks in the `instrumented` however reduced the throughput back to the level of the `vanilla` target. Object clobbering causes a 1% throughput decrease in this experiment.

In the `speedometer` benchmark, which also measures throughput, we observe a 2.0% decrease in throughput from type isolation, and an additional 10.7% decrease from pointer checks, which amounts to a total throughput decrease of 12.7%. Object clobbering accounts for a further 0.5–1% decrease in throughput.

One other metric we can extract from the benchmarks is the average wall clock time per pointer check. For the `speedometer` benchmark, we computed an average difference of 8.5 seconds in total execution time between the `instrument` build and the `vanilla` build. This extra run time is mostly the result of the added pointer fetches as we have observed above. As a lower bound, we can estimate a pointer check rate of 1.1 billion per 8.5 seconds, which corresponds to 7.7 nano-seconds per check, or about 27.8 clock cycles per check. By rough estimation, this indicates that most invocations of `fetchAndCheck` can be served from a CPU cache without a slow DRAM memory access, for example due to one of the the fast paths of the algorithm being applicable.

We attribute the drastic difference between the outcome of the `dromaeo` and `speedometer` benchmarks to their drastically different object usage patterns. While in `dromaeo`, the same or very similar operations involving similar objects are iterated many times in a loop, `speedometer` emulates the behavior of modern, complex JavaScript applications interfacing with the DOM in a realistic way. It appears that the latter usage pattern is affected more by the cache locality and code caching issues that come with type isolation as well as our pointer fetch instrumentation.

4.2.4. Memory usage results

In all benchmarks, we observed from 450-600 distinct types being allocated on IsoHeaps, and 1000-1200 physical pages of memory (around 4 MiB) being occupied by our OFFSET data structure. Note that this memory region is common to each content process and could be located in shared memory, without requiring re-initialization by every new process.

Additionally, in order to measure the memory overhead of the automatic type isolation instrumentation, we polled the `VmRSS` entry of the `/proc/<PID>/status` Linux file to collect the maximum residual memory size of the renderer process across an entire benchmark run. The results of this measurement are not quite conclusive: For `speedometer` benchmark, we measured RSS maxima of 370, 366 and 405 MiB in 3 different runs with the `vanilla` build. For the unchecked build on the same benchmark, we measured RSS maxima of 384, 430 and 389 MiB across three runs. Our best guess is that non-deterministic garbage collection behavior is causing the high variance. The best conclusion that we can draw from this is that while slightly more memory is used by the build with automatic type isolation enabled, it likely

benchmark	target	n	μ	σ	min	max
gmail	vanilla	5	5.03	0.19	4.88	5.14
gmail	unchecked	5	5.22	0.35	5.08	5.46
gmail	unchecked-clobber	5	5.48	0.33	5.35	5.76
gmail	intrumented	5	5.55	0.43	5.28	5.77
gmail	intrumented-clobber	5	6.01	0.37	5.83	6.32
dromaeo	vanilla	5	12752.4	82.81	12709	12808
dromaeo	unchecked	5	13064.0	109.22	13013	13125
dromaeo	unchecked-clobber	5	12979.6	37.62	12963	13007
dromaeo	intrumented	5	12783.4	41.41	12756	12804
dromaeo	intrumented-clobber	5	12631.6	309.99	12413	12841
speedometer	vanilla	40	69.89	7.01	66.28	71.86
speedometer	unchecked	40	68.40	8.01	65.18	70.69
speedometer	unchecked-clobber	40	68.09	9.01	64.98	71.14
speedometer	instrument	40	61.02	5.03	59.52	63.21
speedometer	instrument-clobber	40	60.31	7.37	56.54	62.61

Table 4.1.: Summary of timing results for the different benchmarks. Numbers represent the metrics defined above in subsection 4.2.1. Note that the `gmail` benchmark measures total time, while `dromaeo` and `speedometer` measure throughput. Hence lower numbers are better in the former case, and higher numbers are better in the latter. n is the number of runs, μ is the mean result, σ is the standard derivation. The "min" and "max" columns represent the minimum and maximum result for each benchmark/target combination.

target	.text (in MiB)	.data + .bss (in MiB)
vanilla	67.69	3.12
unchecked	96.96	4.39
unchecked-clobber	97.22	4.39
instrument	110.36	4.42
instrument-clobber	110.62	4.42

Table 4.2.: Summary of code and data size of WebKit libraries.

stays within a 10-30% margin of the uninstrumented build. However, this is not a statistically significant statement. Rigorously proving this would probably require a much more deterministic benchmark.

4.2.5. Binary size results

While we did not optimize for code size or binary size in general during our implementation, it is still an interesting metric to understand, especially due to its indirect effect on cache efficiency. For each of the target builds, we stripped `libwebkit2gtk-4.0.so` of debug symbols and compared the size of the `.text` segment, which contains the source code, as well as the combined of the `.data` and `.bss` segments containing read-only and writable data. This file contains the entire WebKit library, which consists mostly of WebCore code but also some UI and GTK rendering logic.

From the results in Table 4.2 it is obvious that both type isolation as well as our pointer fetch instrumentation add a considerable amount of code. Code size went up by 43.2% due to type isolation, and an additional 19.8% of the original `vanilla` code size due to the added pointer checks. This is most likely the result of extensive use of compile-time constructs such as templates and function inlining, which can potentially be improved by careful refactoring without sacrificing performance – perhaps even improving performance as a side effect. The additional `memset` calls required for clobbering have an almost negligible impact of less than 0.4% on code size.

4.3. Security evaluation

Our conjecture is that the proposed instrumentation provides meaningful security benefits against memory corruption exploits. To investigate this hypothesis, we analyzed it manually as a security mitigation from both a practical and theoretical angle: First, we looked at past, known vulnerabilities in WebCore and categorized them by their underlying vulnerability class. As far as this was viable on a case-by-case basis, we tried to imagine and test how our modifications would have affected the practical exploitability of these vulnerabilities in hindsight.

As a result, we identified general bug patterns and vulnerability classes against

which the proposed approach of automatic type isolation and pointer type checking is particularly effective, and which classes of bugs are generally powerful enough to bypass it. We also considered the mitigation from a theoretical viewpoint, identifying generic weaknesses, some of which are inherent to the approach and some of which could be addressed using additional implementation work, static analysis or instrumentation.

4.3.1. Empirical analysis based on patched vulnerabilities

For our empirical evaluation, we collected all reports of security issues in WebCore published in 2017 and 2018 that were available publicly with at least a very brief analysis such as a reproducing test case and an AddressSanitizer stack trace [14]. A detailed overview can be found in Table A.1. This includes 36 issues reported by the Google Project Zero team which affect WebKit on macOS in the default configuration, 35 of which have an assigned CVE number and for one of which a public exploit is available. Furthermore, we analyzed CVE-2018-4199, exploited and reported by MWR Labs at the Pwn2Own Vancouver competition in March 2018 for which an detailed write-up is available [15], as well as a variant of this bug which fixed in March 2019. MWR Labs was nice enough to also provide us with details of a different WebCore vulnerability, CVE-2019-6233, which they exploited at Pwn2Own Tokyo in November 2018. To complement our data set, we also consider CVE-2018-4121, which is a vulnerability in JavaScriptCore rather than WebCore, but affects the FastMalloc heap which is used by both libraries [16]. It will serve as an example of a very powerful spatial memory corruption vulnerability.

All vulnerabilities are heap-based – affecting objects on the IsoHeap, FastMalloc heap and the heap provided by the C++ standard library – and can be categorized into one of the following classes:

- *uaf-iso*: Use-after-free inside an allocation on an IsoHeap; (17)
- *uaf-object-wild*: Use-after-free inside an allocation on a wild heap; (8)
- *uaf-complex-wild*: Use-after-free inside a complex allocation on a wild heap; (4)
- *oob*: Out-of-bounds read or write on a wild heap; (7)
- *downcast*: Type confusion due to an invalid pointer downcast. (4)

Here, by wild heaps we mean the C++ standard heap and FastMalloc. It has to be pointed out that this distribution is biased for multiple reasons: Most of these issues were discovered using a fuzzer called DOMato [17], which is specifically designed to find DOM-related memory management issues. Furthermore, many of these issues are variants of each other. For example, the above-mentioned bug fixed in March 2019 is an issue closely related to CVE-2018-4199. It is not surprising that it was eventually found due to its similarity. Similarly, CVE-2017-2362, CVE-2017-2460 and CVE-2017-13791 are all issues related to the very same array data structure.

For easier reasoning about exploitability, we assume our attacker has knowledge of the address space of a process. She can achieve this by either exploiting the issue at hand itself as an information leak, or by employing a separate vulnerability such as a pure out-of-bounds read to leak addresses, which we will not discuss in detail here. Essentially, we consider ASLR to be broken as a mitigation beforehand. We will now give some background into how these vulnerability types can often be exploited in practice, and show how in many ways, mistyped pointers or type confusions are an important intermediate goal of the exploit process. This intuition is what drove our core idea of leveraging type information for mitigation purposes in the first place.

Exploitation of *uaf-object-wild* and *uaf-complex-wild* bugs

A total of 11 issues in our data set are use-after-frees of objects or array buffers on a wild heap. A generic way to exploit these types of bugs is to replace the freed object with crafted data. This involves the following steps:

1. Via the vulnerability, cause the affected object or array to be freed while a dangling pointer to it persists in memory;
2. Reclaim the freed space with well-controlled data such as a binary blob or a UTF-16-encoded string;
3. Cause the dangling pointer to be reused to potentially perform a sequence of pointer dereferences and then perform a virtual call.

The second step will effectively turn the vulnerability into a type confusion situation, where the dangling pointer is directed to an object of an unrelated type. In fact a similar situation occurs with *downcast* bugs as described above, however in a strictly less flexible setting: While in the use-after-free case, reclaiming can occur with an arbitrary object type – possibly with certain restrictions such as a fixed object size – in the invalid downcast case the set of target types is restricted at least to other sub-classes of the original pointer type.

In step 3, the attacker fully controls the pointer used to perform the virtual call on. Hence, she can freely forge a virtual table of her liking, which lets her fully control the destination of the virtual call. This yields a primitive where an arbitrary function can be called with an arbitrary argument, which is enough for a full compromise in the vast amount of real-world programs.

This attack has two preconditions: The attacker must be able to perform powerful enough heap manipulations (such as multiple allocations of controlled data) in between the free of an object and the re-use of a dangling pointer to it. Additionally, there must be a virtual call performed on an object reached via references from the reclaimed object which can be initiated by the attacker. In C++, the latter is often trivially true due to the fact that every virtual class has a virtual destructor and hence merely causing an object to get freed results in a virtual call. From a cursory analysis based on the test cases provided by the bug reports as well as those added to the WebKit repository as regression tests, we estimate that at least 10 of the

11 issues in our data set fulfill the second precondition, and at least 9 fulfill both preconditions for this generic attack.

Let us consider the case where a modern mitigation such as fine-grained control flow integrity [3] or CFIXX is employed [11]. In this scenario the third step cannot be used due to explicit instrumentation of virtual call sites, or in other words, the second precondition is not fulfilled. One possibility the attacker still has is to analyze the specific vulnerability at hand and try to mount a so-called *data-only attack*, where instead of hijacking control flow directly, she tries to obtain powerful memory write primitives. This is often possible by forging pointers to other data structures and eventually controlling pointers to which controlled values are written, often in a multi-step process. Since this is a highly non-trivial manual process, we did not analyze the feasibility of this approach for the vulnerabilities in our data set.

We know based on available write-ups and exploits, as well as personal discussion with exploit developers, that at least two of the use-after-free issues in our data set were exploitable using this vtable hijacking approach, and in both cases the same vulnerability was powerful enough to also derive an ASLR-breaking information leak. In at least one case (CVE-2018-4314, discussed in section 4.3.1), this leak was obtained via a data-only memory corruption attack [18]. It is likely that the same holds true for some of the other issues even though we have not analyzed this aspect in depth.

Exploitation of *oob* bugs

In general, exploitation of buffer overflows and over-reads on the heap depends heavily on the environment in which the bug occurs and on its side conditions, such as what data is read or written, at what offset outside of an allocation, and how much control the attacker has over the heap layout and the order of objects on the heap.

A common approach to exploiting buffer overflows is to corrupt live objects on the heap, leading to a situation analogous to the previous section about *uaf-wild* exploitation. Out-of-bounds reads can also lead to memory corruption if structured data such as pointers, indexes or offsets are fetched from attacker-controlled data outside the affected allocation. In this case, exploitation depends heavily on the precise issue at hand. An example is discussed in the case study of CVE-2018-4199. Overall this vulnerability class is very versatile and powerful buffer overflows are generally hard to defend against using memory safety mitigations. Memory tagging is a potential candidate for a hardware-assisted mechanism addressing the root cause of overflows itself, rather than just the symptoms [4].

In our data set we identified two pure buffer overflows, three pure out-of-bounds reads, and two issues where data is fetched as well as written outside allocation boundaries. We estimate that two of the three pure out-of-bounds reads cannot by themselves lead to memory corruption since only string data is read from out-of-bounds, and one of the buffer overflows would be very hard to exploit since the overflow is an unbounded `memmove` operation which will eventually and inevitably

crash the process. This leaves CVE-2018-4199 from Pwn2Own Vancouver 2018 and its variant fixed in March 2019, the JavaScriptCore bug CVE-2018-4121, as well as CVE-2017-13784. For the former two issues detailed write-ups exist published [15, 16].

Exploitability of *uaf-iso* bugs

With the introduction of type isolation using the IsoHeap mechanism for certain object types, primarily DOM nodes, WebKit introduced this limited form of use-after-free. For details of the mechanism please refer to subsection 3.7.1. In the context of memory safety mitigations, we introduced the underlying idea as type-safe memory reuse. It enforces that dangling pointer into such isolated regions will never point to an object of a different type at a later time, since the freed space will never be reused for any other object type. This fully mitigates the vtable hijack attack described above. We see at least two inherent shortcomings of the specific implementation employed in WebKit.

One shortcoming is that in the original WebKit implementation, only type consistency of dangling pointers is enforced, not type safety. Specifically, a dangling pointer can point to a freed object which might still be partially initialized and could still contain dangling pointers to other objects outside the IsoHeap. As an example, consider an object of type T on the IsoHeap which stores a pointer to an array of pointers of type $U*$. The destructor of T deallocates the array, but legitimately leaves the pointer to the array dangling, since language semantics do not guarantee anything about the state of the object past its existence. Usage of a dangling $T*$ pointer can now lead to usage of a dangling pointer $U*$ that points to a memory location outside the IsoHeap, without any type consistency guarantees. In this case, an *uaf-iso* bug can plausibly turn into a more powerful *uaf-object-wild* or *uaf-complex-wild* issue. We have not analyzed our data set in detail to identify issues of this nature, however since only a small percentage of WebCore types is actually placed in IsoHeaps in the original WebKit codebase – less than 400 out of 14257 to be exact – we consider it highly likely that such problematic chains of dereferences exist. Our proposed solution is the clobbering of objects via a simple `memset` after they are freed, and we investigated the performance impact of this additional operation using our `unchecked-clobber` WebKit build in subsection 4.2.3.

Another shortcoming that we identified is that multiple pointers to the same allocation can co-exist and be treated as pointers to different objects. The implications of this situation are highly object-specific and conceivable issues that can occur as a result include the violation of the object state invariants leading to memory corruption issues and race conditions in the case of multi-threaded usage.

We are not aware of any examples of WebCore use-after-free issues involving IsoHeap objects that have been publicly demonstrated to be exploitable. It appears that public researchers have mostly accepted IsoHeap as a strong mitigation and are considering issues in these regions unlikely to be exploitable a priori. This observation is what led us to investigate the actual performance of IsoHeaps in more detail and employ it to a larger extent automatically as part of our instrumentation.

We believe however that at least the information leak part of the published exploit for CVE-2018-4314, a *uaf-object-wild* type issue in detail further below, is powerful enough to be exploitable even if the affected object is moved to an IsoHeap.

Impact of our instrumentation

For purposes of estimating the practical impact of our instrumentation as a memory safety mitigation, we focused on the bug classes other than *downcast*. This decision is based on the fact that eliminating the possibility of these types of errors is very easily achievable by using readily available compiler technologies such as a combination of RTTI[7] and C++ `dynamic_cast`, which are deliberately avoided by the WebKit codebase in the given instances. This easy to employ, universal fix is in our opinion the best solution to the problem, even though we could probably also implement similar checks using our type data structure.

Out of the remaining 36 issues, we deem three as highly unlikely to be exploitable due to being pure out of bounds reads of string data or unbounded buffer overflows. We categorized 17 issues as use-after-frees purely on the IsoHeap, exploitability of which would be mostly unaffected by our instrumentation. One of the reported out-of-bounds reads (CVE-2017-13784) was too intricate for us to understand from the bug report and patch alone and to draw any conclusions regarding its exploitability within a reasonable time frame. This leaves 14 issues for which we believe exploitability to be highly likely, for three of which exploits has even be demonstrated in detail by security researchers. The results of our study into how our instrumentation impacts exploitability are as follows:

- For 7 out of the 8 *uaf-object-wild* issues, the affected objects were automatically put into an IsoHeap, which is the best possible outcome we could hope to achieve with our approach. In the case of the remaining one, CVE-2018-4314, the situation is less clear, and we address this case specifically in the following.
- Three of the four *uaf-complex-wild* issues affect an array data structure which contains only pointers of type `WebCore::FormAssociatedElement*`, which is a type automatically placed in the CHECKED set of types, and thus type safety is ensured for pointers fetched using the dangling reference to the array. The fourth issue (CVE-2018-4089) affects an array of `T` values, where `T` is the template parameter of `WebCore::SVGPropertyTearOff<T>`. According to our analysis of the code base, the only non-primitive types used to instantiate this template are `WTF::String` and `RefPtr<WebCore::SVGPathSeg>`, where the string is only ever read from after object initialization. Since `SVGPathSeg` is in CHECKED, our instrumentation ensures pointer type safety and eliminates at least the exploitation path via the generic vtable hijack method.
- CVE-2018-4199, exploited at Pwn2Own Vancouver 2018 by MWR Labs, as well as a variant of it fixed in March 2019, is an out-of-bounds read of a pointer

[7]Run-time type identification

followed by an out-of-bounds write to the same location. The conditions of this vulnerability are quite complex and our analysis of exploitability under our instrumentation is inconclusive, even though the affected pointer is in the CHECKED set. The instrumentation does however deny the avenue of exploitation that MWR Labs took. We will discuss this case in more detail below.

- The only JavaScriptCore vulnerability in our data set, CVE-2018-4121, a well-controlled heap buffer overflow which was exploited by MWR Labs, is completely unaffected by our instrumentation.

Case study CVE-2018-4199

CVE-2018-4199 is an issue reported, documented, and exploited by MWR Labs [15]. A related issue was fixed in March 2019.[8] The root cause for both bugs is that the element at index -1 is removed from a `WTF::Vector<RefPtr<SVGPathSeg>>`. In pseudocode, this operation amounts to the following steps:

```
// Fetch WTF::Vector backing buffer
RefPtr<SVGPathSeg>* buffer = vector.buffer()
// Call RefPtr destructor (decrementing ref count and if necessary
//   deleting the object)
~RefPtr<SVGPathSeg>(buffer[-1])
// Shift elements to the left
memmove(buffer - 1, buffer, vector.size() - 1)
```

First, a `RefPtr` is fetched from index -1 of the vector, which is then passed to the `RefPtr` destructor. Afterwards all elements are shifted to the left by one position, overwriting the element at index -1 with that at index 0. MWR Labs exploited this vulnerability in two steps, both of which are based on placing a forged pointer at offset -1 of the vector: First they construct an ASLR-breaking information leak by abusing the `RefPtr` destructor's value decrement semantics to corrupt a vtable pointer and call a virtual function of an unrelated class with a `SVGPathSeg` object as the **`this`** argument. Afterwards they hijack control flow by forging a `SVGPathSeg` object with reference count one and causing its virtual destructor to be invoked.

Both steps are not possible in our instrumented build due to the pointer type safety property, since `SVGPathSeg` is in the CHECKED set of types. To verify that this is the case, we took the regression test case provided in the above-mentioned WebKit patch and put a break point on the affected call to `removeItemFromList`:

[8] `https://github.com/WebKit/webkit/commit/c6936191317d`

```
(gdb) br SVGAnimatedPathSegListPropertyTearOff::removeItemFromList
Breakpoint 4 at 0x7f003c029e10 (2 locations)
(gdb) c
Continuing.

Thread 1 "WebKitWebProces" hit Breakpoint 4, 0x00007f003c029e10 in
  WebCore::SVGAnimatedPathSegListPropertyTearOff::
    removeItemFromList(unsigned long, bool)@plt ()
   from /home/niklas/typeiso/webkit/WebKitBuild[...]
(gdb) i r rdi rsi
rdi            0x304490010070   53071031894128
rsi            0xffffffffffffffff       -1
(gdb) p ((WebCore::SVGAnimatedPathSegListPropertyTearOff*)
            $rdi)->m_values.m_buffer
$5 = (WTF::RefPtr<
        WebCore::SVGPathSeg,
        WTF::DumbPtrTraits<WebCore::SVGPathSeg> > *)0x381c0014cd80
(gdb) x/6gx 0x381c0014cd80-8
0x381c0014cd78: 0x0000381c077f2090 0x00003047c8020120
0x381c0014cd88: 0x00003047580200c0 0x0000000000000000
(gdb) c
Continuing.

Thread 1 "WebKitWebProces" received signal SIGILL,
    Illegal instruction.
bmalloc::fetchCheckedImpl<2277ul, 1ul> (p=0x381c0014cd78) at
    ../../Source/bmalloc/bmalloc/VMAllocateInlines.h:267
267                 BAD();
(gdb) x/1i $rip
=> 0x7f003ecf1ad0 <WTF::Vector<WTF::RefPtr<WebCore::SVGPathSeg,
                   WTF::DumbPtrTraits<WebCore::SVGPathSeg> >,
                     0ul, WTF::CrashOnOverflow,
                     16ul>::remove(unsigned long)+288>: ud2
```

In this case we can see that there is a valid address (`0x0000381c077f2090`) at index -1, however it is not consistent with the type `SVGPathSeg`, hence the pointer checking algorithm stops the program. One way to try and exploit this issue while avoiding pointer type unsafety is to focus on the out-of-bounds write which is following the out-of-bounds read in the form of a `memmove` operation. Surviving the call to the `RefPtr` destructor is easy, we can just place a zero value in front of the array. It will then get overwritten by the pointer which was previously the first entry in the array. This leads to the following primitive: Overwrite a zero value located at the end of an allocation with a pointer to a `SVGPathSeg`. Another possible approach is to place a pointer to a live `SVGPathSeg` object at index -1 and to free it by using the vulnerability. This leads to a situation analogous to *uaf-iso*. It is unclear if a full exploit can be constructed from these primitives alone.

Case study CVE-2018-4314

A detailed write-up of this *uaf-object-wild* vulnerability and its exploitation was published by the discoverer [18]. The exploit has two phases: In the first phase, a `SVGAnimatedTypeAnimator` object is freed but not reclaimed, and a dangling pointer to the still intact object is used to violate the object state invariants of a `SVGAnimateElementBase` object, leading to a powerful type confusion situation. A mistyped pointer is then used to write a controlled address, leading to a disclosure of heap memory at that address. This phase is used to obtain a pointer into the code segment of the Safari binary and break ASLR.

In a second phase, the generic vtable hijack exploitation technique described in section 4.3.1 is used to redirect control flow and achieve code execution. We observe that due to the fact that all subclasses of `SVGAnimatedTypeAnimator` are allocated on IsoHeaps in our instrumented build, this control flow hijack is not viable. However, since the first phase of the exploit does not violate type safety, our instrumentation does not affect it, except for the fact that with object clobbering upon free, a new object of the same type needs to be allocated to reclaim the space of the old one – a trivial adaption.

One might be inclined to call this a positive outcome. Nevertheless, we cannot neglect the possibility that the same type confusion used for the information leak part of the exploit could also be used for more powerful memory corruption if the conditions are right. From out analysis it does not appear as if any dangerous memory corruption result in controlled writes can occur given the type confusion primitive. However, since this bug is complex and multi-faceted, this can not be excluded with full certainty.

4.3.2. Challenging the implementation

In this section we will discuss weaknesses of our specific implementation of pointer type safety and how they could potentially be addressed in the future using additional implementation work or hardware support.

As already discussed in section 4.3.1, currently we do not instrument pointer casts and thus we cannot detect invalid casts leading to pointer type confusions. We do not think out-of-band type information is the correct way to address this issue. However, our type data structure could in theory be used instead of in-band type information such as RTTI, which is based on virtual tables. This would require detecting and instrumenting sites where a pointer is cast to a type in CHECKED.

Another weakness of our specific implementation results from the usage of the original WebKit IsoHeap implementation, which uses *bmalloc* as the allocator algorithm. bmalloc maintains a bitmap tracking the allocated slots inside an `IsoPage`, a 16 KiB heap arena in which objects are allocated. Since this bitmap is also part of the same heap region, it is conceivable that an attacker with very good control over the allocation pattern of a specific object type could forge near-arbitrary data inside this bitmap. While this would be a very hard to achieve deterministically, it

is worth considering to move these bitmaps to a separate memory region. This could however induce a performance penalty from decreased cache locality.

In section 3.2 we introduced a relaxed version of type pointer safety which we called wild pointer type safety. In early experiments we learned that this is an important optimization to keep the performance overhead within reasonable limits. The intuition that it follows is that *oob* bugs are unlikely to occur if only constant-size objects are involved, an assumption which was confirmed by our data set. However, there is at least one rather common coding pattern that allows for the occurrence of out-of-bounds issues even in this settings: *Small buffer optimization* is a technique where array-like data structures are stored inline of a constant-size type only if they are smaller than a certain threshold. For example, in the WebKit code base, `WTF::Vector` has a template parameter that defaults to 0 and describes the number of elements that can be stored inside the object without allocating out-of-line heap memory. If the number of elements is small enough, the pointer stored inside the vector object which would normally point to a different heap allocation, in fact points inside the vector itself. An out-of-bounds write in this scenario could be used to corrupt the vector object and surrounding data. In fact, memory tagging when used as a generic mitigation is affected by the same problem, unless the implementation of the small buffer optimization code specifically provides additional information to the allocator. We could use the same information to mark regions as wild where fixed-size inline arrays can occur, however we need it to be available at compile time in order to perform the proper static analysis.

5. Conclusions

We presented an out-of-band type metadata data structure for C++ programs, based on the concept of type isolation. Our prototype implementation makes use of a two-phase Clang compiler instrumentation and was successfully applied to the WebKit code base as a case study.

Our analysis of real-world vulnerabilities has demonstrated that our instrumentation can be used as a security mitigation and that it prevents or considerably increases the difficulty of exploitation of memory corruption issues in many instances. Both automatic type isolation as well as our newly introduced pointer type safety has meaningful impact on real-world vulnerabilities in the primary subject of our research, the WebCore library. There are multiple examples of easily exploitable bugs, for which the obvious exploitation routes were rendered ineffective. In particular, in multiple cases classic use-after-free scenarios on a mixed-type heap and arbitrary pointer dereferences were respectively turned into much more restricted use-after-frees on the IsoHeap and type-safe pointer dereferences.

In some more complex cases we analyzed the underlying vulnerabilities in detail and elaborated on the exact memory corruption primitives obtained with and without our instrumentation, showing that the primitives are considerably weaker with than without the pointer type safety policy employed. However, disproving exploitability is generally a very hard task. We firmly believe that in some of the considered cases exploitation is still possible.

We observe that type isolation and type safe pointers can be especially effective to mitigate various kinds of use-after-free bugs as well as out-of-bounds reads on the heap. It is unlikely to be effective against powerful heap buffer overflow vulnerabilities, due to the fact that with this primitive at hand, an attacker has a lot of freedom to corrupt uninstrumented pointers or non-pointer control data such as length or index fields, which allows for the construction of increasingly powerful primitives, often leading to near-arbitrary memory reads and writes.

However, it is obvious from our experiments that the performance impact induced by the extent to which we applied our instrumentation could be prohibitive in the context of software with high performance requirements such as a web browser. After all, performance is often the main reason why a memory unsafe language such as C++ is chosen in the first place. In this regard, it appears that automatic type isolation using static analysis, potentially with object clobbering upon free, has a much more favorable trade-off between the performance overhead and security benefits compared to our implementation of pointer type safety. There is no inherent reason why type isolation per se should add considerable overhead – even though we could measure a slight difference in performance in our experiments – while adding type checks in

many places is clearly an expensive instrumentation. Out of the 16 issues in our data set which are proven to be, or which we consider likely to be exploitable, only 5 were mainly impacted by pointer type safety rather than by the type isolation part of our instrumentation.

5.1. Future work

We have learned from our case study that out-of-bounds writes are often powerful enough to bypass the security features of our instrumentation entirely, for example by corrupting uninstrumented pointers in the same heap region. In our current implementation, exactly one wild heap is used where all non-IsoHeap allocations occur. One heuristic we propose is to separate the wild heap into two regions: In one region, we make sure that no allocations contain uninstrumented pointers. Out-of-bounds writes in this region would be harder to exploit because no pointers can be corrupted without pointer type safety being violated. All other types are placed in a different heap region. However, it seems likely that in complex programs, even the former heap can contain critical control data structures such as offset or length values which could lead to more powerful, near-arbitrary memory writes.

The most obvious disadvantage of our proposal is the heavy performance penalty which the instrumentation of pointer fetches introduces. One possible solution here is to investigate the usage of hardware features such as ARM v8.3 *pointer authentication codes* [19]. Since our allocator ensures type-safe memory reuse, a pointer signed to be valid for a specific type will remain safe to use in the future in conformance with our pointer type safety policy. Our pointer fetch algorithm would just be a hardware pointer authentication operation. This shifts most of the instrumentation overhead from the pointer fetch sites to the pointer store sites, where pointers potentially have to be authenticated and then re-signed for a different type during type casts.

Bibliography

[1] J. Bialek, "The Evolution of CFI Attacks and Defensess." https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf, 2018.

[2] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *2015 IEEE Symposium on Security and Privacy*, pp. 745–762, May 2015.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 4:1–4:40, Nov. 2009.

[4] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018.

[5] https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a .

[6] V. A. Dinakar Dhurjati, Sumant Kowshik and C. Lattner, "Memory Safety Without Runtime Checks or Garbage Collection," in *Proc. Languages Compilers and Tools for Embedded Systems 2003*, (San Diego, CA), June 2003.

[7] P. Akritidis, "Cling: A Memory Allocator to Mitigate Dangling Pointers," in *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2010.

[8] B. Lee, C. Song, T. Kim, and W. Lee, "Type Casting Verification: Stopping an Emerging Attack Vector," in *USENIX Security Symposium*, pp. 81–96, USENIX Association, 2015.

[9] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 517–528, ACM, 2016.

[10] G. J. Duck and R. H. C. Yap, "Effectivesan: Type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 181–195, ACM, 2018.

[11] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++ virtual dispatch," in *Prof. of ISOC Network & Distributed System Security Symposium (NDSS)*, 2018.

[12] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and Generic Metadata Management with Mid-Fat Pointers," in *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, (New York, NY, USA), pp. 9:1–9:6, ACM, 2017.

[13] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos, "METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening," in *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, (New York, NY, USA), pp. 5:1–5:6, ACM, 2016.

[14] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.

[15] MWR Labs. `https://labs.mwrinfosecurity.com/assets/BlogFiles/apple-safari-pwn2own-vuln-write-up-2018-10-29-final.pdf`.

[16] MWR Labs. `https://labs.mwrinfosecurity.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf`.

[17] I. Fratric. `https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html`.

[18] I. Fratric. `https://googleprojectzero.blogspot.com/2018/10/365-days-later-finding-and-exploiting.html`.

[19] `https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf`.

Erklärung

Hiermit erkläre ich, Niklas Baumstark, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Appendix

See next page.

Bug class	CVE Identifier	Bug ID	Details	Fix	Exploitability	Impact TI	Impact TI + PC
uaf-complex-wild	2017-2362	165959	P0 1044	be3dfa8f931	likely	no	yes
uaf-complex-wild	2017-2460	167200	P0 1090	ec8df6820d1	likely	no	yes
uaf-complex-wild	2017-13791	176368	P0 1355	d0ac97f994f	likely	no	yes
uaf-complex-wild	2018-4089	180745	P0 1477	unknown	likely	no	no
uaf-object-wild	2018-4314	186658	P0 1596	2b1eab35917	proven	yes	yes
uaf-object-wild	2018-6233 or 6234	191661	MWR Pwn2Own	5648ddc1c72	proven	yes	yes
uaf-object-wild	2017-2476	167885	P0 1114	bc488af86aa	likely	yes	yes
uaf-object-wild	2017-7042	171376	P0 1244	c03c585e658	likely	yes	yes
uaf-object-wild	2017-13792	176160	P0 1345	a178814f417	likely	yes	yes
uaf-object-wild	2017-13794	176364	P0 1353	01be685af14	likely	yes	yes
uaf-object-wild	2017-13802	176224	P0 1351	98ae9624c45	unknown	yes	yes
uaf-object-wild	2018-4317	186657	P0 1595	20e325c1445	unknown	yes	yes
oob (read+write)	2018-4199	unknown	writeup[a]	unknown	proven	no	partial
oob (read+write)	unknown	195333	variant of above	c6936191317	proven	no	partial
oob (write)	2018-4121	unknown	writeup[b]	c6deeea41e5	proven	no	no
oob (read)	2017-13784	176220	P0 1349	6a6af9b1466	unknown	unknown	unknown
oob (write)	2017-2459	167162	P0 1087	74777ae4dc4	unlikely	unknown	unknown
oob (read)	2017-13785	176219	P0 1348	6cad78f472c	unlikely	unknown	unknown
oob (read)	2018-4328	187251	P0 1610	280ebf6e98a	unlikely	unknown	unknown
uaf-iso	2017-2454	167092	P0 1080	e52df7b1bae	unknown	no	no
uaf-iso	2017-2455	167117	P0 1082	68a59060090	unknown	no	no
uaf-iso	2017-2466	167310	P0 1097	8efa0b43e62	unknown	no	no
uaf-iso	2017-2471	167502	P0 1105	b516c1596c0	unknown	no	no
uaf-iso	2017-7039	171373	P0 1241	436b7228d37	unknown	no	no
uaf-iso	2017-7040	171374	P0 1242	0163afe53d2	unknown	no	no
uaf-iso	2017-7041	171375	P0 1243	90a7b239e51	unknown	no	no
uaf-iso	2017-13796	176159	P0 1344	cb76b01b59d	unknown	no	no
uaf-iso	2017-13797	176161	P0 1346	unknown	unknown	no	no
uaf-iso	2017-13798	176367	P0 1354	2a298098ee8	unknown	no	no
uaf-iso	no CVE	180525	P0 1465	bd5b1f97007	unknown	no	no
uaf-iso	2018-4200	182383	P0 1525	8b37be52e6e	unknown	no	no
uaf-iso	2018-4197	186655	P0 1593	65d4dbca722	unknown	no	no
uaf-iso	2018-4318	186656	P0 1594	b475f64aab9	unknown	no	no
uaf-iso	2018-4306	186917	P0 1602	unknown	unknown	no	no
uaf-iso	2018-4315	186925	P0 1604	98dd2fb0e64	unknown	no	no
uaf-iso	2018-4323	187249	P0 1609	b10f9026aa7	unknown	no	no
downcast	2017-2373	184032	P0 1038	36b0eaec6f9	unknown	no	no
downcast	2017-2369	165145	P0 999	aaced4cec6a	unknown	no	no
downcast	2017-7049	171547	P0 1250	8b54864637a	unknown	no	no
downcast	2017-13783	176221	P0 1350	0231e8adc46	unknown	no	no

Table A.1.: Summary of all considered WebCore and JavaScriptCore bugs. The bug IDs refer to the public WebKit bug tracker at `https://bugs.webkit.org/`. Bugs marked as *P0 <XXXX>* refer to the Project Zero bug tracker at `https://bugs.chromium.org/p/project-zero/issues/detail?id=XXXX`. The "Fix" column refers to the Git commit in the `https://github.com/WebKit/webkit/` mirror repository that fixes the given issue. In some cases it was not possible to determine the CVE identifier, bug ID or fixing commit of an issue, which is indicated by an "unknown" note. The column "Impact TI" describes whether automatic type isolation has an impact on exploitability of the corresponding issue, while "Impact TI + PC" describes the impact of type isolation combined with the introduced pointer type safety checks.

[a] `https://labs.mwrinfosecurity.com/assets/BlogFiles/apple-safari-pwn2own-vuln-write-up-2018-10-29-final.pdf`
[b] `https://labs.mwrinfosecurity.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf`