The Undefined Quest for Full Memory Safety

by

Ronald Gil

B.S., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author . . . .
Department of Electrical Engineering and Computer Science
December 22, 2017

Certified by . . . .
Howard E. Shrobe
Principal Research Scientist, CSAIL
Thesis Supervisor

Certified by . . . .
Hamed Okhravi
Senior Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

The Undefined Quest for Full Memory Safety

by

Ronald Gil

Submitted to the Department of Electrical Engineering and Computer Science
on December 22, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we explore full memory safety and the various intricacies involved. We analyze existing memory safety techniques in both hardware and software and their many different goals. This task involves determining the limits of the protections guaranteed by these different protection systems, regardless of whether they were explicitly or implicitly stated. It is demonstrated that the common software technique of protecting only allocation bounds does not provide nearly enough of a barrier for attackers. Then, we go beyond particular schemes and examine the limitations of languages, C in particular. We discover many corner cases and ambiguities that prevent even the best possible protection system from providing full memory safety in the context of the C language specification. We also collect some results for the prevalence of these issues, present approaches to further analyze them, and consider how they might extend into other languages or systems.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist, CSAIL

Thesis Supervisor: Hamed Okhravi
Title: Senior Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to thank my thesis supervisors Dr. Howard Shrobe and Dr. Hamed Okhravi for their guidance throughout the completion of this work. It would not have been possible without their continued insight and support.

I would also like to thank Dr. Stelios Sidiroglou-Douskos, Dr. Robert Watson, and others with whom I had enlightening conversations about various topics involved in this work.

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory corruption is a classic problem that has plagued many systems and served as a reliable source of exploits over the years [55]. Low level languages, such as C and C++, which allow direct control of memory provide ample room for introducing vulnerabilities through specification ambiguities, permitted unsafe usage, and their general flexibility [44,46,47]. In fact, some of these issues make any system that honors the language specification incapable of providing complete memory safety [11].

1.1 Memory Safety

There is a large amount of literature about providing memory safety [55]. However, the guarantees provided by a full memory safe system are not necessarily well-defined. Many systems have tried to provide only partial safety with the goal of increasing the difficulty for exploit development. This goal also applies to commonly deployed protection schemes such as ASLR and W$\oplus$X. ASLR attempts to make it harder to obtain library addresses by randomizing their location in memory. W$\oplus$X instead focuses on avoiding code injection by preventing any writeable data from being executed as code.

In general, memory safety issues can be classified as spatial or temporal. To provide complete memory safety, a system should be able to provide both complete spatial and temporal safety.

1.1.1 Spatial

Spatial memory safety involves preventing memory accesses from going out of their intended bounds. For example, a pointer to an array should not be able to access memory outside of that array as shown in Figure 1-1. In some cases, the context of a pointer or access is critical in determining the correct bounds.

```
typedef struct {
  ...
  char foo[32];
  ...
} mystruct;

mystruct s;
char* arr = s.foo;

char a = arr[-8]; // Invalid
char b = arr[0];  // Valid
char c = arr[16]; // Valid
char d = arr[40]; // Invalid
```

Figure 1-1: An example of accesses that should be valid or invalid to guarantee spatial safety.

1.1.2 Temporal

Temporal memory safety involves preventing memory accesses to an incorrect or invalid object in the context of object allocation and deallocation. For example, a pointer to an object should not allow access to the underlying memory after the object has been deallocated (commonly referred to as use after free). This idea is shown in Figure 1-2.

1.2 Defense Systems

It is common for defense systems to focus only on either spatial safety or temporal safety [15,24,34,35]. Furthermore, there have been hardware-only, software-only, and

```
char* ptr = malloc(64);

for (int i=0; i < 64; i++) {
  ptr[i] = '0'; // Valid
}
ptr[20] = 'a';  // Valid
ptr[42] = 'b';  // Valid
ptr[30] == 'c'; // Valid

free(ptr);

for (int i=0; i < 64; i++) {
  ptr[i] = '1'; // Invalid
}
ptr[20] = 'x';  // Invalid
ptr[42] = 'y';  // Invalid
ptr[30] == 'z'; // Invalid
```

Figure 1-2: An example of accesses that should be valid or invalid to guarantee temporal safety.

hybrid designs. Each choice has its own pros and cons, but only a hybrid solution can independently provide complete memory safety across the entire stack.

1.2.1 Hardware Systems

On the hardware side, defense systems have attempted to make safety guarantees about execution and memory in a variety of ways. For example, one main approach is to tag memory with metadata that can be used to verify correctness in some way [17, 18, 40, 59, 64]. Even within this grouping, tagged architectures, there is a divide between approaches that choose to tag all data in memory [18] and those that tag only certain parts such as pointers [59]. There are other key dimensions that distinguish tagged architectures from each other including policy type, compatibility, and trusted computing base (TCB).

There are performance benefits to providing memory safety through hardware, but there are also limits on expressiveness and dynamic policy control. Both of these aspects are better handled through software which can use the mechanisms provided by hardware to create higher-level policies.

1.2.2 Software Systems

By comparison, software techniques generally suffer from greater performance overhead [2, 26, 34, 35, 62]. However, they make it easier to maintain backwards compatibility and are simpler to deploy.

Without hardware support for memory safety, software schemes can at best try to encapsulate their protections in a way that isolates their interaction with inherently unsafe hardware interaction. Current systems architecture makes this complicated since the kernel, which serves as the hardware interaction layer, tends to have the greatest privileges of any software subsystem.

1.3 Limitations of C

However, even a system that intends to provide full memory safety regardless of the overhead will be limited by the language in which it operates. For example, C has an abundance of edge cases that are difficult or impossible to secure if the system intends to follow the C standard.

1.4 Beyond C

While completely securing C in an automated fashion is impossible, corner cases that similarly limit what protection can be provided also exist in other languages. Thus, a system that intends to provide full memory safety needs to carefully consider its software base and its limitations.

Chapter 2

Hardware Defenses

2.1 Overview

There have been a variety of proposals that rely on hardware in order to provide memory safety [18, 24, 28, 53, 60, 61]. Some of these are pure hardware schemes while others provide different levels of integration with the software stack. Hardware mechanisms can achieve better performance than software mechanisms but also lack the level of control offered by software mechanisms. Furthermore, low-level mechanisms can sometimes make it difficult to express more abstract guarantees. For example, a hardware system may provide coarse-grained protections of memory regions, but it may not be clear in the context of the software stack how it can provide more general guarantees or policies. Overall, there has been a trend favoring hybrid hardware-software approaches over pure hardware approaches in order to exploit the benefits of each type of approach.

2.2 Tagged Architectures

Tagged machines date back to at least the early 70s [37], however there has been an increasing interest in recent times. The core idea is that some or all parts of memory have an associated tag which provides extra information about the piece that can be used when verifying validity. There are many different aspects to this idea, such

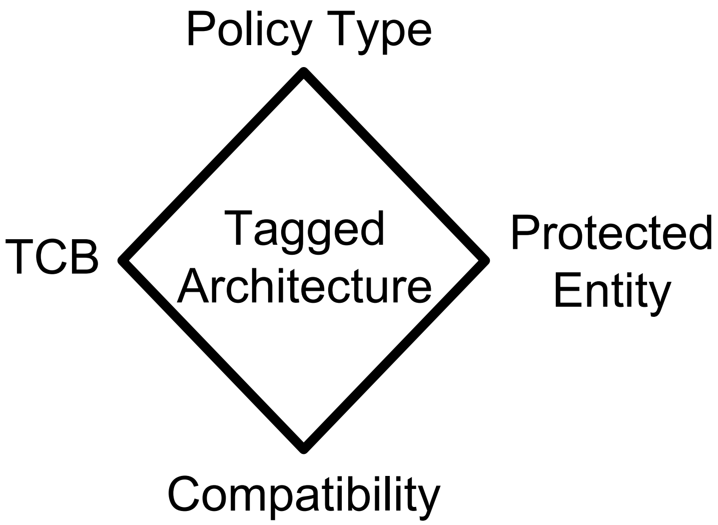


Figure 2-1: Dimensions of a tagged architecture design.

as the granularity of the tags, how many extra bits of information are stored, what words are tagged, how are tags verified, *etc.* These design choices directly impact the performance and compatibility of these architectures.

2.2.1 Dimensions

There are many details that vary between different tagged architectures, but a smaller classification based on four features gives a good overall idea of their distinctions. These four features are policy type, TCB, protected entity, and compatibility. Table 2.1 shows the corresponding features and evaluation method for tagged architectures over the years.

Policy Type

A policy serves as the key component in providing security guarantees by defining what operations are allowed or disallowed. The granularity of the filter definition can be broad or specific depending on the particular policy. Architectures may choose to directly provide mechanisms to provide certain policies or delegate the task to

Table 2.1: Tagged architectures with their design dimensions and evaluation methods sorted by the publication year.

Architecture	Year	Policy Type	Protected Entity	Software Compatibility Goals	Evaluation
Dover [54]	2017	Configurable	Data	Source	FPGA
HDFI [51]	2016	Isolation	Data	Full	FPGA
Taxi [22,23]	2015	Memory Safety	References	Source	Simulator
PUMP [16,18]	2015	Configurable	Data	None	Simulator
CHERI [58,59,61]	2014	Configurable	References	Full	FPGA
SPARC M7 SSM [40,43]	2014	Memory Safety	References	Full	Processor
WatchdogLite [32,33]	2014	Memory Safety	References	Full	Simulator
Intel MPX [24]	2013	Memory Safety: Spatial	References	Full	Processor
Low-Fat Pointers [28]	2013	Memory Safety	References	None	FPGA
SAFE [4,17]	2012	Configurable	Data	None	FPGA
DataSafe [10]	2012	Isolation	Data	Full	Simulator
Harmoni [14]	2012	Configurable	Data	None	FPGA
Shioya, *et al.* [48]	2011	Isolation	Data	None	Simulator
SIFT [39]	2011	Configurable	Data	Full	Processor Core
TIARA [49]	2009	Configurable	Data	None	Theoretical
DIFT Coprocessor [27]	2009	Configurable	Data	Full	FPGA
HardBound [15]	2008	Memory Safety: Spatial	References	Full	Simulator
Loki [63]	2008	Isolation	References	Full	FPGA
FlexiTaint [57]	2008	Configurable	Data	Full	Simulator
SECTAG [3]	2007	Configurable	Data	Full	Simulator
Raksha [13]	2007	Configurable	Data	Full	FPGA
SecureBit2 [41]	2006	Memory Safety: Inter-Process	References	Full	Emulator
Minos [12]	2004	Memory Safety	Data	Full	Emulator
DIFT [53]	2004	Memory Safety	Data	Full	Simulator
RIFLE [56]	2004	Isolation	Data	Source	Software Estimation
AEGIS [52]	2003	Isolation	Data	None	Simulator
Mondriaan [60]	2002	Configurable	Data	Full	Simulator
Aries [7]	2001	Isolation	Data	None	Theoretical
XOM [29]	2000	Isolation	Data	Full	Simulator
M-Machine [8]	1994	Isolation	References	None	Whole System
KCM [5]	1989	Memory Safety	Data	Source	Processor
SPUR [64]	1987	Memory Safety	Data	None	Simulator
Lisp Machine [31]	1985	Memory Safety	Data	None	Whole System
HEP [50]	1982	Isolation	Data	None	Whole System
Burroughs [38]	1973	Memory Safety	Data	None	Whole System

software.

The types of policies can be memory safety policies, isolation policies, or configurable as some combination of both. The way these policies are implemented varies greatly between different tagged architectures. For example, isolation is sometimes provided through information flow control (IFC) with a corresponding controlled communication mechanism. In other cases, memory safety can be used as a way to provide isolation.

Trusted Computing Base (TCB)

A trusted computing base (TCB) can be critical for managing and configuring tags in a tagged architecture. The particular components in the TCB determine the impact that bugs in different parts of the system will have on the safety guarantees provided by the whole system. The consequences can range from completely voiding all guarantees to having no effect. The TCB is often an integral part of a tagged architecture's design and can be directly influenced by other design choices (e.g. integration with the MMU).

Table 2.2 shows a compilation of the architectures with their corresponding TCB. A component is not considered to be part of the TCB if the guarantees provided by the architecture remain valid when that component is compromised. Furthermore, designs which only partially include certain components (e.g. the kernel) in their TCB are listed as having that component in their TCB.

Protected Entity

Tagged architectures choose to protect either data or references. For data protection schemes, tags follow all words through registers and are used to control where words are stored. Reference protection schemes instead use tags to identify and extend pointers. These two approaches are an important distinction as they provide different protection properties and overheads. Some schemes make use of such properties as a basis for things such as rights delegation. The overhead for data protection schemes scales with the amount of data while for reference protection schemes it scales with the amount of references.

Table 2.2: Tagged architectures with their TCB sorted by the publication year.

Architecture	**Year**	**Tag Generation Compiler**	**Tag Enforcement Compiler**	**Loader**	**MMU**	**Processor**	**Kernel**
Dover [54]	2017	Yes	Yes	Yes	No	Yes	No
HDFI [51]	2016	No	No	No	Yes	Yes	No
Taxi [22,23]	2015	No	No	Yes	Yes	Yes	Yes
PUMP [16,18]	2015	No	No	No	No	Yes	No
CHERI [58,59,61]	2014	No	No	No	No	Yes	No
SPARC M7 SSM [40,43]	2014	No	No	No	No	Yes	No
WatchdogLite [32,33]	2014	No	No	Yes	Yes	Yes	Yes
Intel MPX [24]	2013	Yes	Yes	Yes	No	Yes	No
Low-Fat Pointers [28]	2013	No	No	No	Yes	Yes	No
SAFE [4,17]	2012	No	No	No	No	Yes	No
DataSafe [10]	2012	No	No	No	Yes	Yes	No
Harmoni [14]	2012	No	No	Yes	Yes	Yes	Yes
Shioya, *et al.* [48]	2011	No	No	Yes	Yes	Yes	Yes
SIFT [39]	2011	No	No	Yes	Yes	Yes	Yes
TIARA [49]	2009	No	No	No	No	Yes	No
DIFT Coprocessor [27]	2009	No	No	No	No	Yes	No
HardBound [15]	2008	Yes	Yes	Yes	Yes	Yes	Yes
Loki [63]	2008	No	No	No	No	Yes	Yes
FlexiTaint [57]	2008	Yes	Yes	Yes	Yes	Yes	Yes
SECTAG [3]	2007	Yes	Yes	Yes	Yes	Yes	Yes
Raksha [13]	2007	No	No	No	No	Yes	Yes
SecureBit2 [41]	2006	No	No	Yes	No	Yes	Yes
Minos [12]	2004	No	No	Yes	No	Yes	Yes
DIFT [53]	2004	Yes	Yes	Yes	No	Yes	Yes
RIFLE [56]	2004	Yes	Yes	Yes	No	Yes	Yes
AEGIS [52]	2003	No	No	No	No	Yes	No
Mondriaan [60]	2002	No	No	No	No	Yes	Yes
Aries [7]	2001	No	No	No	Yes	Yes	Yes
XOM [29]	2000	No	No	No	No	Yes	No
M-Machine [8]	1994	No	No	Yes	No	Yes	No
KCM [5]	1989	No	No	No	Yes	Yes	No
SPUR [64]	1987	Yes	Yes	Yes	Yes	Yes	Yes
Lisp Machine [31]	1985	Yes	Yes	Yes	Yes	Yes	Yes
HEP [50]	1982	Yes	Yes	Yes	Yes	Yes	Yes
Burroughs [38]	1973	Yes	Yes	Yes	Yes	Yes	Yes

Compatibility

Tagged architectures aim for different levels of compatibility with existing code and systems as part of their design.

In terms of software this choice means providing some degree of source code and/or binary compatibility. In some cases these compatibilities are achieved by requiring modifications elsewhere on the software stack or optionally in source code. For example, certain kernel modifications may be able to prevent spurious fault detections in user space programs, or annotations may be optionally required to enable protections afforded by the tagged architecture. Software structures impose limitations that force a trade between security and compatibility. Consequently, architectures must choose if and how their provided mechanisms account for these issues when dealing with software compatibility.

The other aspect tagged architectures consider is hardware compatibilities. There are various pieces of hardware that architectures may either modify or interact with (e.g. the MMU). Generally, architectures must be clear in their design about how they interact with the processor and main memory with respect to tags. However, the amount of consideration for other components varies based on how the design is evaluated. For example, purely theoretical schemes may not fully consider certain hardware interactions while prototyped designs are forced to explicitly consider interaction with every physical component.

2.2.2 Challenges

Tagged architectures face a variety of challenges while trying to provide security guarantees. Interestingly, some of these issues (e.g. edge cases and overheads) are pertinent for software protection schemes and are discussed in that context in later sections.

Tag Guarantees

A critical part of tagged architectures are the tags themselves. Architectures need to consider how they can guarantee tag correctness and authenticity. These requirements are handled in a variety of ways for different designs. For instance, capability based designs provide the concept of capabilities which are consequently derived from one another in a clear chain.

Dynamic Linking and Dynamic Loading

Dynamic linking and dynamic loading are widely used in a wide variety of applications. They ease tasks such as code sharing or reuse (e.g. for libraries) and loading components at runtime (e.g. plugins). However, these features can also create policy enforcement conflicts. For example, if a library is compiled with a different policy from the application it is used in, it can introduce disparities in what the enforcement mechanism should do. Thus, architectures can choose to handle this issue as part of their design or delegate it to the application level by requiring that every component be compiled with the same or compatible policies.

Overheads

There are various overheads that tagged architectures must consider such as memory, performance, silicon, and power. Memory and performance overheads are highly dependent on the particular design of how tags are stored and processed. They may even scale differently depending on characteristics such as whether an architecture protects data or references. These different overheads can also impose limitations on whether a tagged system can feasibly run in different contexts. For example, mobile and embedded devices may be unable to use a tagged system due to the extra power usage introduced by a tagged architecture.

Heterogeneous architectures

Most modern processors have accelerators. However, there has not been much work done in figuring out how systems work with such third party IP components. A primary exception is WHISK [42], a DIFT architecture that attempts to address the issue. Many other details in the context of heterogeneous architectures such as multiprocessor coherence and consistency can also use further analysis.

Dynamic Code Generation

Dynamic code generation techniques (e.g. JIT) introduce their own set of questions that architecture designs would need to consider such as what generates the tags and how integration with tools might work. This topic is another area that has not been well explored in the context of tagged architectures.

Distributed Systems

In order to properly function within the context of a distributed system, storage and transfer of tag information becomes crucial. For example, designs would need to examine the implications, if any, of tags being stored in nonvolatile storage. There would also need to be consideration given to how tag information may be transferred across a network. Furthermore, policy enforcement across processes, hosts, and other groups introduces its own set of complexities and challenges.

Side Channels

Side channel attacks such as timing or storage attacks can potentially break isolation guarantees. Systems need to take care to ensure that their isolation policies, especially those involving IFC, can prevent untrusted code from leaking sensitive data through these means.

Bounds Checking Precision

Determining precise bounds for arbitrary applications can become difficult due to numerous factors such as language ambiguity or missing information. The conservative approach would be to make bounds as restrictive as possible in every scenario, but it comes at the expense of sometimes breaking systems and specifications. At the other end of the spectrum, bounds could be chosen to be the least restrictive of all possible choices. Yet, this approach can then break security guarantees. When dealing with these problematic scenarios, many tagged architectures choose to expose an option to dial the behavior instead of making it an inherent part of the design.

Precise bounds checking is crucial to providing safety guarantees as exploits have been designed around even off by one overflows. Some systems attempt to address this precision issue in the context of specific language specifications and even then there are often complications that arise (e.g. dealing with custom memory allocators, language ambiguities, etc.). However, there are further edge cases that need to be considered in other contexts such as assembly functions and signal handlers.

2.3 Memory Management Units (MMUs)

Memory management units (MMUs) are widely deployed in existing systems today. They provide an abstraction layer by maintaining a mapping of virtual memory to physical memory with associated permissions per page.

The MMU is commonly used to provide isolation. For example, the kernel commonly uses virtual addressing to isolate different processes. It can further be used to provide memory safety at the page level (e.g. by using guard pages). However, page sizes tend to be relatively large (commonly 4KB), which makes protecting smaller objects expensive. This coarse-grained protection is useful only when used conservatively as a sandboxing mechanism due to the potentially steep performance penalties.

Due to their widespread use, most schemes tend to address how they interact with the MMU if at all. Some designs choose to fully integrate with it and preserve the existing process model while others simply discard it. In either case, the overhead imposed by virtual addressing serves as a useful baseline for different protection schemes.

Chapter 3

Software Defenses

3.1 Overview

Pure software schemes trade off performance in favor of easier deployability and sometimes compatibility. While there are schemes that provide complete protection (within the limitations of the language) [34,35], the performance overhead is exceedingly high and thus there has been no noticeable adoption of them in practice. Furthermore, in order to obtain the complete memory protection guarantees these schemes often incur false positives.

Other schemes have instead opted to provide partial protection in an effort to achieve low overhead. One strategy that is sometimes used, which can greatly reduce overhead, is to protect only allocation bounds rather than sub-object bounds.

3.2 Allocation Bounds

An allocation bounds scheme sets and checks bounds based on the size of an allocated object. Thus, members of composite objects (e.g. structs) have the same bounds as the allocated containing object. The result of this distinction is that inter-object corruption is prevented, but not intra-object corruption.

3.2.1 Examples

There have been a number of previous designs that offer to protect only allocation bounds.

Low Fat

The original implementation of the low-fat pointer scheme was introduced in hardware [28]. This hardware low-fat scheme provides fine-grained spatial safety while reducing the overhead of fat pointers. It uses 18 bits of a 64-bit word to contain a block size, lowest valid multiple of the block size, and largest valid multiple of the block size. The other 46 bits are used to store the pointer address. These four values are then used to reconstruct the base and bounds. Whenever a computed pointer goes out of bounds, it is permanently changed to be an `Out-of-Bounds Pointer` hardware type. This type of pointer will produce an error if there is ever an attempt to dereference it.

It is important to note that due to the dependence on a fixed block size, this scheme by default offers only an approximation, albeit a relatively accurate one, to the actual bounds. This problem can be remedied by having the compiler pad sub-objects, so that every sub-object is aligned to the exact block size. Moreover, since the bounds can be set independently of the pointer address, a pointer can be narrowed to a sub-object's bounds.

Similar to its hardware counterpart, the recently proposed low-fat pointer software schemes rely on bits stored in the 64-bit pointer representation to determine bounds on stack [21] or heap [20]. However, instead of storing separate bits and reducing the representable address space, it encodes the bounds into the address itself. For this setup to work, pointers are assigned to specific address ranges based on the size of the object they point to as shown in Figure 3-1. Then, pointer arithmetic is instrumented to check the size corresponding to a particular region. Given the size of a region and the fact that all objects in a region are of equal size, the code can then safely determine if pointers resulting from pointer arithmetic are outside of these bounds.

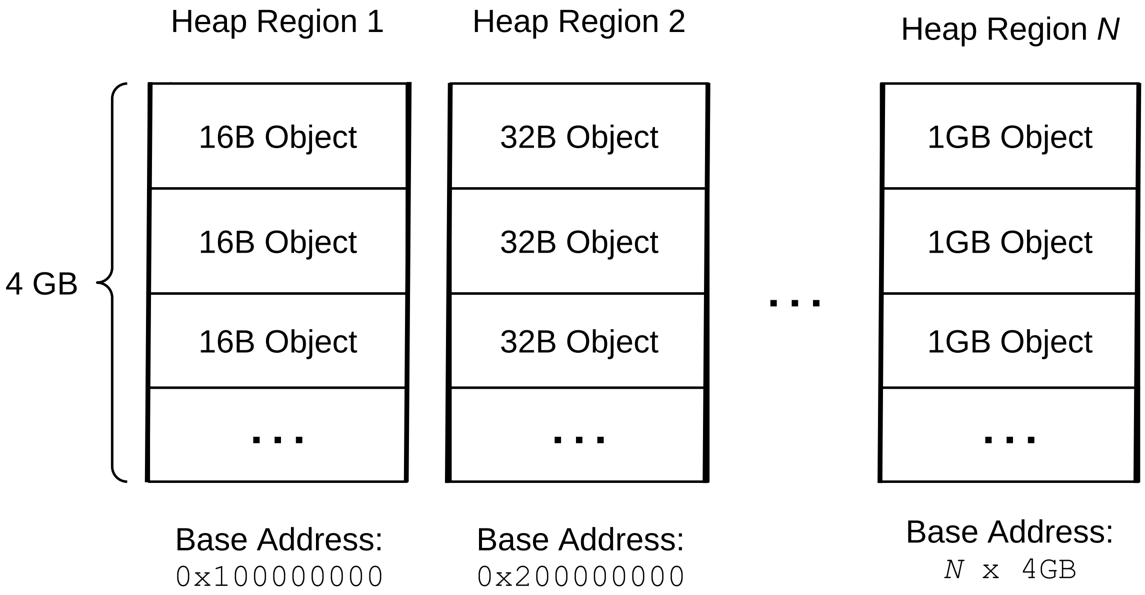


Figure 3-1: An overview of how heap regions are designated in the low-fat software scheme to store objects of a particular size.

Consequently, this strategy encodes the bounds information quite efficiently and reduces he runtime overhead of safety checks. However, this efficiency comes at a cost to precision compared to the hardware scheme. Because there is a correlation between a pointer's value and its bounds, there is no inherent ability to narrow bounds for sub-objects. Hence, the software scheme only protects allocation bounds.

Baggy Bounds

Baggy Bounds [2] functions like the low-fat pointer schemes in that it tracks object sizes based on bits in the pointer address. However, it uses these bits to index into an external table stored in memory to determine the actual object bounds for a pointer. Every pointer arithmetic and array indexing operation is then instrumented to check that the pointer after the operation lies within the bounds set by the base and size of the original pointer. Thus, Baggy Bounds stores information both in the pointer address as well as an external structure just like the low-fat software scheme.

PAriCheck

PAriCheck [62] takes a different approach and instead keeps a table of numeric labels for all allocated memory regions. Then, it compares the labels of the memory regions referenced by a pointer before and after each pointer arithmetic operation to determine if it has gone out of bounds. While PAriCheck does not rely on storing any information in the pointer address itself, it still needs an external structure that is accessed for every check. Furthermore, it carries the risk that the fixed number of labels available (based on label size) can run out.

Earlier Designs

Even earlier attempts which tracked referent objects such as Jones and Kelly's GCC patch (J&K) [26] and other schemes directly based on it [19,45] also provided at most allocation bounds protection. The difficulty and overhead of modifying these designs to protect object bounds varies but none originally attempted to do so as presented.

3.2.2 Limitations

Unfortunately, while making it slightly more difficult to mount an attack, allocation bounds protection still leaves a program exposed to relatively common attack vectors. Since all sub-objects in a composite object share the same bounds as the parent object, any pointer to a composite object can freely read and write anywhere within the parent object. In other words, overflows are still possible as long as they remain within the bounds of the parent object.

```c
typedef struct {
        char buf[124];
        void *fptr;
} mystruct;

mystruct s;
```

Figure 3-2: A struct vulnerable to intra-object corruption.

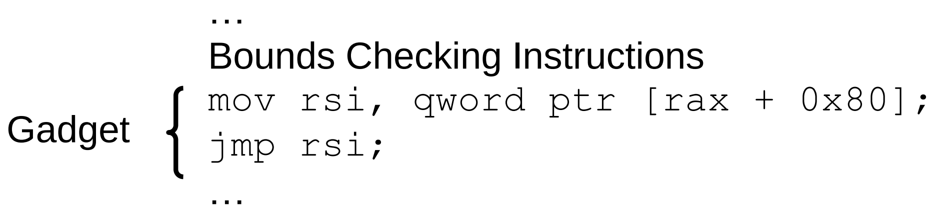
One example of a problematic setup would therefore be a struct with both an array and a function pointer as shown in Figure 3-2. An attacker can use a standard buffer overflow to overwrite the function pointer and hijack control. Moreover, this particular setup is relatively common in major C programs as shown in Table 3.1.

	Definitions	Array Accesses	Function Pointer Accesses	Other Accesses	Total Accesses
git	8	140	51	965	1156
httpd	7	68	15	842	925
lighttpd	1	0	7	468	475
nginx	3	55	60	2789	2904
openssl	6	6	217	2674	2897
postgresql	8	1	26	709	736
redis	1	0	5	10	15
acl	0	0	0	0	0
bash	2	3	14	76	93
coreutils	1	6	2	19	27
e2fsprogs	6	13	235	5446	5694
findutils	0	0	0	0	0
gcc	1	0	2	62	64
grep	3	10	6	90	106
gzip	0	0	0	0	0
hostname	0	0	0	0	0
ncurses	2	68	37	1158	1263
pam	1	0	1	33	34
perl	0	0	0	0	0
sed	1	0	0	0	0
slang	8	103	103	953	1159
tar	0	0	0	0	0
util-linux	1	0	3	24	27
zlib	0	0	0	0	0

Table 3.1: Existence and usage of structs with both function pointers and arrays in common C programs.

There are various other attacks that remain possible within the limits imposed by the allocation bounds protection scheme such as data oriented attacks.

Since both the stack-based and the heap-based low-fat schemes focus on spatial memory safety, we demonstrate the pointer stretching attack, which uses spatial corruption, as a concrete example.



```
...
Bounds Checking Instructions
mov rsi, qword ptr [rax + 0x80];
jmp rsi;
...
```

Figure 3-3: Selecting a gadget to avoid including bounds checks.

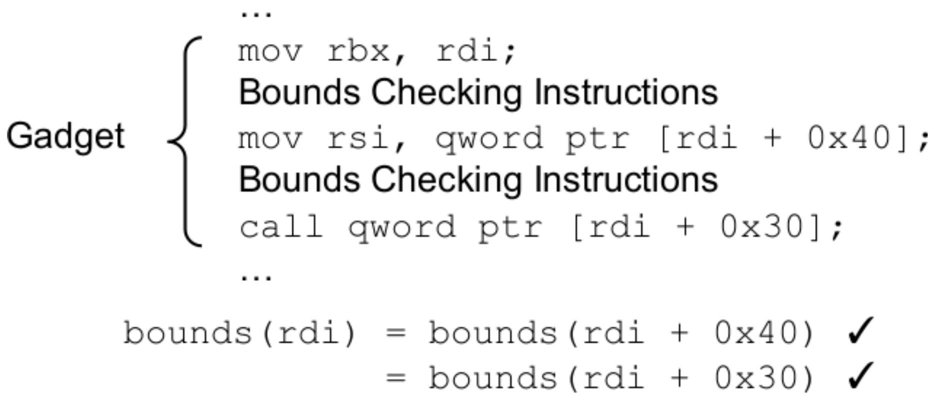
Pointer Stretching

The first step in a pointer stretching attack[1] is creating an intra-object corruption. Consider the code snippet in Figure 3-2, and assume the `struct` is allocated on the stack or heap. Since the attack uses only intra-object corruption, it can be assumed that the allocation bounds corresponding to the `struct` are exact. In other words, inter-object corruption is assumed to be impossible. In this case, the `struct` is stored in a region with the designated size of 128 bytes that matches its size perfectly. Any pointer to any field of this `struct` is inherently pointing to the same region where the `struct` is stored, thus it can point to any field of the `struct` even when the protection is enabled. For example, the bounds of `*s.buf` encompass the entire `struct`, while it should legitimately only point to `buf`. As a result, we can overflow the buffer inside `mystruct` to control the function pointer `*ftpr`. Since this modification of the function pointer is done through an overflow, and not the intrinsic instructions in the application itself, it is not instrumented by the low-fat schemes. As a result, after the overflow, `*ftpr` can point to any region in memory.

Selecting the gadgets themselves from a hardened binary is not problematic. In most cases, the start of a gadget can be selected to avoid including bounds checking instructions as shown in Figure 3-3. When the checks cannot be avoided, such as for consecutive memory accesses in one gadget, the only restriction imposed is that the bounds on the start address for any indexing done in the instruction must correspond to the bounds of the resulting address. This scenario is shown in Figure 3-4. The exploits later presented use both cases.

Although it may seem like such an overwrite can be dangerous, what we have at

[1] The name refers to the fact that buffer pointer bounds are "stretched" to corrupt other sub-objects stored in a composite data structure.



```
...
mov rbx, rdi;
Bounds Checking Instructions
mov rsi, qword ptr [rdi + 0x40];
Bounds Checking Instructions
call qword ptr [rdi + 0x30];
...

bounds(rdi) = bounds(rdi + 0x40) ✓
            = bounds(rdi + 0x30) ✓
```

Figure 3-4: An out of bounds error can be avoided by ensuring accesses in a gadget do not go out of bounds when used.

this point is far from a complete attack. The overwritten function pointer can point to a ROP gadget, but for an attack to succeed, we need to be able to run a series of ROP gadgets. Even if the goal is to ultimately run one gadget that issues a system call, for example to launch a shell, we need to be able to set the arguments (%rax, %rdi, %rsi, etc.) properly, which necessitates a chaining mechanism. However, we do not currently control the stack or have a trampoline to chain gadgets together. Moreover, it is unlikely to find a gadget that loads the stack pointer (%rsp) from an area currently under our control (the `struct`). To overcome these challenges, we leverage a trick we call *gadget hopping*.

3.2.3 Gadget Hopping

It is possible to find a series of gadgets that ultimately modify %rsp to point to the area under our control (stack pivoting), but we need to initially chain these gadgets somehow without relying on the stack. In previous code reuse attacks, control always returns to a central location that contains a list of gadget addresses. This central location is the stack in ROP attacks and a trampoline in JOP attacks [6, 9]. We observe that this central location is not necessary, particularly for short sequences of gadgets. By carefully selecting gadgets, we can come up with a set of gadgets in a way that each gadget directly transfers control to the next one either through a call or a jump, until the last gadget properly pivots the stack. We call this trick

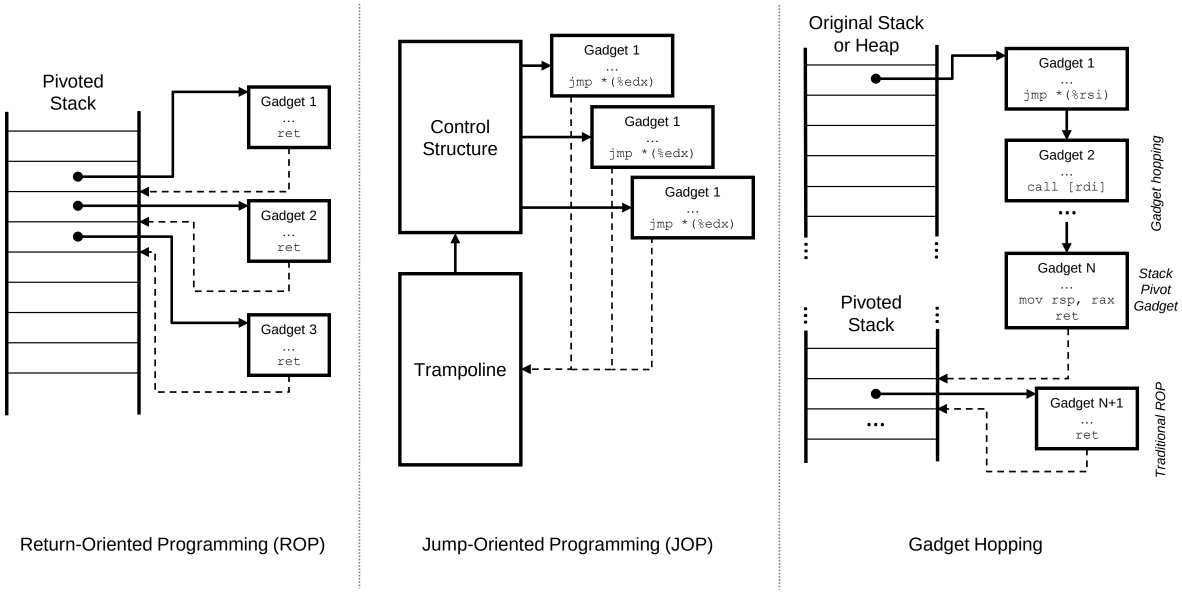


Figure 3-5: Gadget hopping compared to traditional ROP and JOP techniques.

gadget hopping that can be used for *delayed pivoting*. Figure 3-5 illustrates this technique and its comparison with ROP and JOP. In traditional ROP attacks, such a technique is not necessary since the attacker controls the stack and a simple `pop rsp;` gadget can be used for pivoting. However, when the system is protected by allocation protection techniques, gadget hopping is necessary since the area under the control of the attacker is initially very small (*e.g.*, the inside of a `struct`).

Consider the following example where we can initially control the inside of a `struct` but not the stack. There is no gadget in *libc* that can load the `%rsp` from the area under our control (which resides on heap). However, there are two gadgets in `libc` listed in Figure 3-6. In this case, `%rax` points to a word inside the `struct` under our control. By maliciously modifying the word pointed to by `%rax` to contain the address of the next gadget minus `0x56` (`0x381A`), we can make the first gadget jump directly to the second one. The second gadget pops the value from the top of the stack, which was the original `%rax` content, to `%rsp`, thus pivoting the stack to the area under our control. Figure 3-7 illustrates this example. Note that the pivoted stack now resides in the struct itself, which is also the site of the initial memory corruption. This is not strictly necessary since the pivoted stack can reside

```
0x0000000000173dc4: push rax; cmp ebp, esi; jmp qword ptr [rax + 0x56];

0x0000000000003870: pop rsp; ret;
```

Figure 3-6: Example gadgets useful for gadget hopping.

in any area under attacker's control. Moreover, note that the pivoted stack will grow upwards, hence the overlap with the `struct`.

The sample exploits demonstrate a more complicated example of *gadget hopping* with hops through three gadgets. After the *delayed pivot* achieved through *gadget hopping*, the attack proceeds as a traditional ROP attack by chaining gadgets together to setup the arguments properly and issue a system call. By executing a system call with control of the arguments, attackers can achieve arbitrarily malicious behavior such as launching a shell, creating a backdoor socket, and so on.

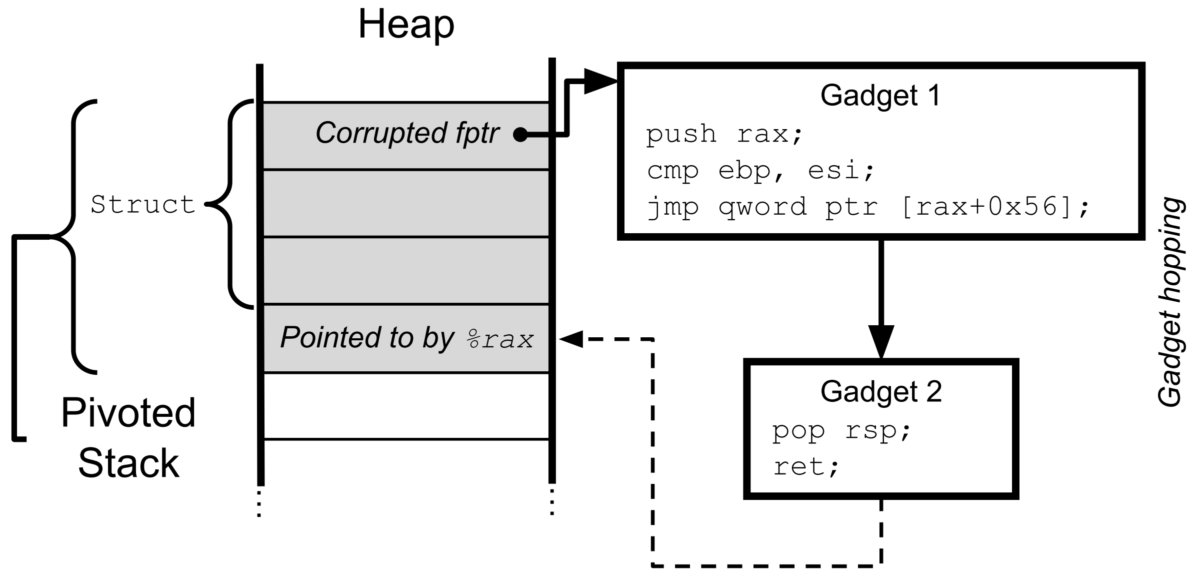


Figure 3-7: Gadget hopping example.

3.2.4 Sample Exploits

To further concretize the vulnerability of allocation bounds schemes, exploits that assume the complete protection of the Low-Fat Pointer scheme [20,21] (an allocation bounds scheme) are presented.

Nginx Heap-based Exploit

The objective of this attack is to achieve arbitrary code execution using the pointer stretching attack and demonstrate that allocation protection is an insufficient granularity for bounds checking. We search for a structure containing both a buffer and a function pointer we can corrupt to ultimately execute a shell. After inspecting the source code, we find the structure `ngx_http_request_s`, shown in Figure 3-8. It is allocated for each request and has both of these features: a buffer, `lowcase_header`, which contains part of the header as it is parsed and a function pointer, `log_handler`, which is called if there is an issue processing the request.

```
struct ngx_http_request_s {
    uint32_t                  signature;  /* "HTTP" */
    ngx_connection_t          connection;
    ...
    ngx_http_log_handler_pt  log_handler;
    ...
    u_char    lowcase_header[NGX_HTTP_LC_HEADER_LEN];
    ...
    unsigned                  http_minor:16;
    unsigned                  http_major:16;
};
```

Figure 3-8: A snippet of the definition of the `ngx_http_request_s` struct.

The buffer is located below the function pointer in the structure, so overwriting it requires a buffer underflow. Similar memory bugs have been discovered in Nginx in the past, for example `CVE-2009-2629` [1]; we assume such a memory bug exists.

The underlying weakness arises from the fact that any pointer to any sub-field of the `ngx_http_request_s` can point to the entire `struct`, thus such a buffer underflow is not prevented by the software-based low-fat schemes as long as the corruption is contained in the `struct` and does not corrupt the areas outside of it. This is illustrated in Figure 3-9. The solid black lines show the correct bounds for an array pointer, the solid gray lines indicate the allowable bounds based on the allocation region, and the dotted lines show out-of-bound accesses that are stopped. Note that the corruption happens purely based on intra-object (`struct`) overwrites, so the function pointer `log_handler` can be corrupted without causing out-of-bound violations.

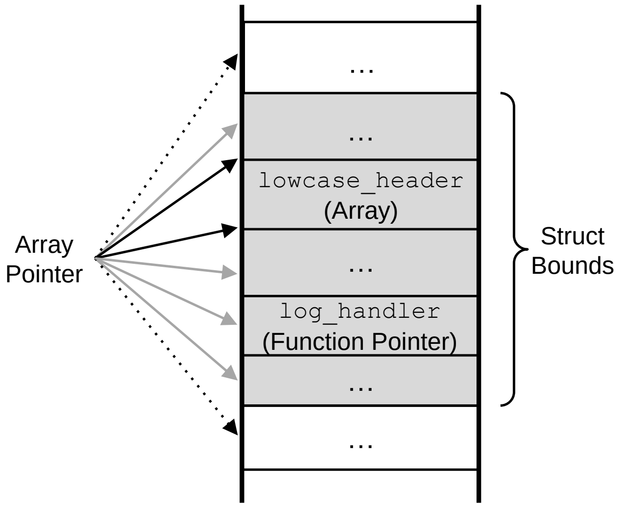


Figure 3-9: A display of the heap region where the vulnerable struct, `ngx_http_request_s`, is allocated and what is accessible given a pointer to any member of the structure (solid arrows).

First, we observe when the corruptible function gets called. Whenever there is any issue processing a request, a worker process calls `ngx_http_log_error` shown in Figure 3-10. Towards the end of this function, the corruptible pointer `log_handler` is invoked. To ensure a problem is detected and control is eventually passed to this function, we simply request a nonexistent page.

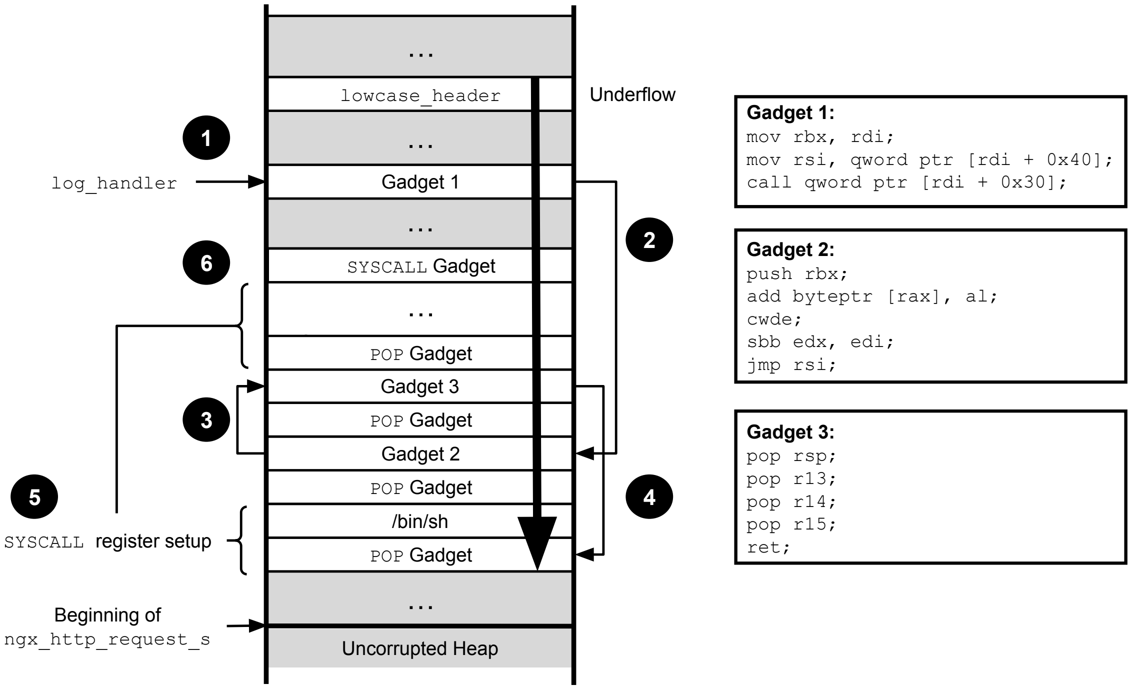
Now, for the initial step of the attack we overwrite the function pointer `log_handler` with a ROP gadget to do a stack pivot. We need the stack pointer to point to a region of memory we control, so that we can continue executing further instructions. Analyzing the call site reveals that there is a pointer to the same struct we are underflowing. This is the variable `r` in Figure 3-10. At the default optimization level, this pointer ends up stored in a register. Furthermore, it gets passed as the first and second argument to `log_handler` since `ctx->current_request` is an alias to the same pointer. Per x86_64-bit calling convention this setup means registers `rdi` and `rsi` contain pointers to this structure which we control.

Unfortunately, there are no available single stack pivot gadgets for the 64-bit registers we can use here. Nevertheless, we can create a *delayed pivot* by *gadget hopping*. As described earlier, the idea is to create a chain of gadgets that call or

```
static u_char * ngx_http_log_error(
   ngx_log_t *log, u_char *buf, size_t len) {
   u_char              *p;
   ngx_http_request_t  *r;
   ngx_http_log_ctx_t  *ctx;

   if (log->action) {
       p = ngx_snprintf(buf, len, " while %s",
           log->action);
       len -= p - buf;
       buf = p;
   }

   ctx = log->data;

   p = ngx_snprintf(buf, len, ", client: %V",
       &ctx->connection->addr_text);
   len -= p - buf;

   r = ctx->request;

   if (r) {
       return r->log_handler(
           r, ctx->current_request, p, len);

   } else {
       p = ngx_snprintf(p, len, ", server: %V",
           &ctx->connection->listening->addr_text);
   }

   return p;
}
```

Figure 3-10: The log error function where the corruptible `log_handler` gets called.

jump to addresses set by previous gadgets in the chain. Only the last gadget ends with a `ret` instruction, by which point we have completed the pivot. More concretely, here we can use a chain of three gadgets to achieve the delayed pivot. The first gadget runs three important instructions: move `rdi` (the struct pointer) into `rbx`, move the contents at address `rdi+0x40` into `rsi`, and call the address stored at `rdi+0x30`. Thus, we set `log_handler` to the first gadget, `rdi+0x30` to the second gadget, and `rdi+0x40` to the third gadget as shown in Figure 3-11. Since `rdi` is simply a pointer to the start of the struct, we use the same original underflow to set all these addresses. Given our setup, after the first gadget executes, the instruction pointer moves to the address stored at `rdi+0x30`, which is the address of the second gadget. The relevant



Gadget 1:

```
mov rbx, rdi;
mov rsi, qword ptr [rdi + 0x40];
call qword ptr [rdi + 0x30];
```

Gadget 2:

```
push rbx;
add byteptr [rax], al;
cwde;
sbb edx, edi;
jmp rsi;
```

Gadget 3:

```
pop rsp;
pop r13;
pop r14;
pop r15;
ret;
```

Figure 3-11: A general overview of the attack and how the stack looks after the stack pivot (i.e. part of the overwritten struct). The `POP` and `SYSCALL` gadget entries refer to a gadgets of the form `INSTRUCTION; ret;`.

instructions in this gadget are pushing `rbx` onto the stack and then jumping to `rsi`. Since the first gadget set `rsi` to the address at `rdi+0x40`, it now points to the address of the third gadget. This gadget completes the stack pivot by popping `rsp` and then popping three other arbitrary data registers. The sole purpose of popping the extra registers is to move the stack pointer beyond the very beginning of the struct in order to preserve the `connection` member of the struct. This data pointer is used between the location of the underflow and the corrupted call site, so it is easier to simply preserve the value for this exploit.

With the stack pivot complete, we can proceed to execute many more gadgets from the new stack owing to the large size of the struct. In particular, we create a chain of gadgets that sets the arguments for an `execve` system call. In order to do complete the system call though, we need to the base address of the `libc` library to obtain a gadget for the final `syscall` instruction. We obtain this address by using a known libc pointer on the stack. Since we know the offset of this pointer from the base pointer, we add it to the address in the `rbp` register to get the address of the libc pointer on the stack. Then, we simply make the address provided for the syscall gadget relative to the known address. Note that libc itself is also "hardened" by the low-fat bounds, but those have no impact on our usage of the gadgets, as described earlier.

At this point, anything currently in the data registers is irrelevant going forward, so we reuse some registers used for the stack pivot. Moreover, for simplicity, the remaining system call setup is done through gadgets of the form instruction; ret; and we refer to them only by the corresponding instruction. First, we pop off the next value on the (pivoted) stack into register r10. Through the underflow, we set this value to be the string "/bin/sh". Then, we use a gadget to set another register, rdi, to the address of an arbitrary valid object in memory. The only constraint is that it needs to be large enough to hold two pointers. Once we have some valid address in the register, we use a gadget to store the pointer in r10 into the address of rdi. We can use the same gadgets to store a null byte in the address of rdi+8, but since inserting a null byte in the underflow is problematic, we xor a register with itself so that it gets zeroed and execute a mov to the correct address.

Finally, we use further pop gadgets to set the actual arguments to the syscall instruction. This includes setting the execve syscall number of 0x3b in rax, setting rdi to the "/bin/sh" string, rsi to the data pointer that contains the string pointer and NULL, and rdx to that pointer + 8 (so it points to NULL).

We bootstrap the exploit by sending the malicious payload to cause the underflow and consequently overwrite the various parts of the struct. Upon completion, the process image switches to a shell. Through a similar construction, we can execute any arbitrary command restricted only by the permissions given to the process (which depends on the web server configuration).

Apache Stack-based Exploit

In this attack, we use pointer stretching to obtain arbitrary code execution in Apache. In a similar fashion to the Nginx attack, we use a struct containing a buffer and a function pointer. However, the struct is located on the stack in this case.

```
typedef struct {
  /* internal state of the scanner */
  const char   *inputbuf;
  int          inputlen;
  ...
  char         scan_buf[MAX_STRING_LEN];
  ...
  /*
  * The function to use to lookup provider
  * functions for variables and funtctions
  */
  ap_expr_lookup_fn_t *lookup_fn;
} ap_expr_parse_ctx_t;
```

Figure 3-12: A snippet of the definition of the `ap_expr_parse_ctx_t` struct.

The attack vector for this exploit is the `httpd` configuration file. An attacker using this vector can, for example, host the malicious configuration file on a code sharing website and advertise its useful features. Unbeknownst to the target, the configuration file then causes the intra-object corruption and hijacks control of the web server.

In order to parse parts of the configuration file, a struct to contain the scanner and parser state, `ap_expr_parse_ctx_t`, is pushed onto the stack. `ap_expr_parse_ctx_t` contains two relevant members: a buffer, `scan_buf`, to store the raw characters being scanned and a function pointer, `lookup_fn`, optionally specified by a module for its configuration parsing (Figure 3-12). The function pointer is always called in the function `ap_expr_info_make` (Figure 3-13), except that it is set to a no-op function when creating the scanner and parser state struct if one was not provided.

The overview of the Apache attack is shown in Figure 3-14. We begin by triggering an overflow of the buffer through a crafted directive in the configuration file. Through this overflow we set the function pointer `lookup_fn` to a simple gadget of `ret 0x1189;`. This instruction moves the stack pointer up into the same buffer that was overflowed[2]. Besides the overflow, we also have the crafted string trigger the `ap_expr_info_make` function by having some basic valid subexpression parsed after

[2] Due to the large size of the buffer, there are many other analogous gadgets that move the stack pointer within the buffer's bounds.

```
static ap_expr_t *ap_expr_info_make(int type, const char *name,
                                    ap_expr_parse_ctx_t *ctx,
                                    const ap_expr_t *arg)
{
    ap_expr_t *info = apr_palloc(ctx->pool, sizeof(ap_expr_t));
    ap_expr_lookup_parms parms;
    parms.type  = type;
    parms.flags = ctx->flags;
    parms.pool  = ctx->pool;
    parms.ptemp = ctx->ptemp;
    parms.name  = name;
    parms.func  = &info->node_arg1;
    parms.data  = &info->node_arg2;
    parms.err   = &ctx->error2;
    parms.arg   = NULL;
    if (arg) {
        switch(arg->node_op) {
            case op_String:
                parms.arg = arg->node_arg1;
                break;
            case op_ListElement:
                do {
                    const ap_expr_t *val = arg->node_arg1;
                    if (val->node_op == op_String) {
                        parms.arg = val->node_arg1;
                    }
                    arg = arg->node_arg2;
                } while (arg != NULL);
                break;
            default:
                break;
        }
    }
    if (ctx->lookup_fn(&parms) != OK)
        return NULL;
    return info;
}
```

Figure 3-13: The ap_expr_info_make function in which lookup_fn gets called.

the lookup_fn pointer has been overwritten. Thereby, execution proceeds at the stack pivot gadget which we set lookup_fn to, and we can execute further instructions by setting appropriate gadgets in the buffer.

Thus, using the same overflow, we set up a chain of a gadgets starting from

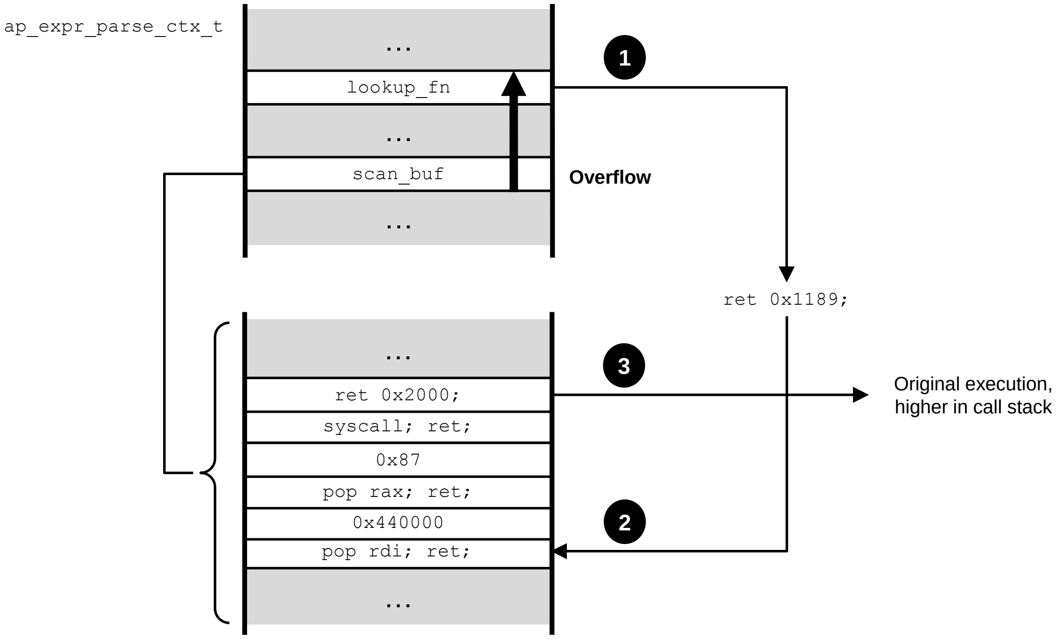


Figure 3-14: An overview of the setup of the struct being exploited and the internals of the buffer setup to execute the `personality` syscall.

the new stack pointer address inside the buffer. We use the gadgets to set up the registers for a system call. Similar to the Nginx exploit, we could trigger `execve` with proper arguments to spawn a shell, but in order to avoid repetition, we give the Apache exploit a different flavor. In particular, we use the `personality` syscall[3], which sets a process' execution domain, to disable W⊕X protection and ASLR. We set the first gadget to be a `pop rdi;` instruction and the address above it to be `0x440000`. This value corresponds to the bitwise OR of the flags `ADDR_NO_RANDOMIZE` and `READ_IMPLIES_EXEC` for a `persona`. They effectively disable ASLR and W⊕X for any new pages mmaped by the process, respectively.

Since the `persona` is the only argument to the `personality` syscall, the remaining gadgets are for executing the system call. To accomplish this we use a `pop` gadget to set `rax` to `0x87`, the syscall number for `personality`, and a gadget with the `syscall` instruction to execute the actual system call. For the syscall gadget we need the address of `libc`, but we use the same strategy as in the previous exploit to obtain it.

[3] This system call is specific to Linux, but the attack generalizes such that any arbitrary code can be run by setting the appropriate gadgets in the overflowed buffer.

At this point, the `persona` of the main process has been changed and execution resumes at the next address in the buffer. However, this result would prove pointless if the server crashed right after it was done since we have just disabled the ASLR and W $\oplus$ X protections. Hence, to avoid a crash we use one last `ret` gadget to move `rsp` to a valid address, say the return of one of the functions on the call stack preceding this one. We could have alternatively used more gadgets if we were unable to be as specific as we needed with the address. Still, the particular address is not important as long as execution proceeds normally thereafter. Note that the danger of such an attack is partly due to its stealthiness. Going forward the server runs normally except with ASLR and W$\oplus$X protections disabled, enabling simple code injection using the buffer overflow described earlier.

3.2.5 Mitigations

There are countermeasures that can be implemented to avoid the setup used in the presented exploits. However, they can have steep performance and memory overheads.

The immediate solution would be to protect sub-object bounds. However, how this would be handled in some designs is unclear. For instance, the low-fat pointer scheme makes bounds highly dependent on addresses, thereby making narrowing complicated. Furthermore, it's likely that most if not all the performance benefits of the design compared to existing sub-object protection schemes will be lost.

Alternatively, a combination of schemes can be used. In other words, composite objects would rely on a scheme that protects sub-object bounds while other objects would use the allocation bounds scheme. However, this mixing of designs can greatly increase implementation complexity and make it difficult to prove safety guarantees.

A more targeted approach may instead choose to allocate sub-objects separately and then maintain a pointer to them in the original object. In this way, the allocation bounds will be correct for every object. Some of the problems with this approach are that it reduces locality and can increases memory overhead since a sub-object is essentially taking up twice the space (the parent object still has to have its full size). As a way to improve upon this idea, instead only vulnerable members such as arrays can be separately allocated. Nonetheless, the same problems still apply and this design overall still doesn't completely prevent intra-object corruption in the case of composite objects. For example, unless all members are also moved, a pointer to somewhere within the composite object remains free to move between members that are not separately allocated.

It's possible to imagine other improvements such as only storing metadata separately for sub-objects, but it is unclear if modifications to protect sub-objects can preserve the performance benefits allocation bound schemes try to achieve.

3.3 Sub-object Bounds

Designs that protect sub-object bounds maintain separate bounds for sub-objects rather than using the allocation bounds of the parent. These schemes include SoftBound [34] with CETS [35] and the hardware low-fat pointer scheme [28] among others.

3.3.1 Limitations

While schemes that protect sub-object bounds avoid the pitfalls of allocation bound schemes, they still cannot provide complete protection without violating the C specification due to ambiguities introduced in the language specification. This issue is further discussed in Section 4.

Chapter 4

Full Memory Safety

4.1 Overview

SoftBound [34] combined with CETS [35] is formally proven to provide spatial and temporal safety (*i.e.* what may be considered full memory safety). However, providing truly complete memory safety requires breaking compatibility at various layers and redesigning parts of existing architectures or software stacks. The reason is that the completeness of a fully memory safe system is limited by the existing software and hardware stack.

4.2 Limits of Hardware

Currently deployed hardware architectures do not inherently support operations to provide full memory safety. There are some features, such as virtual addressing provided by the MMU, which have limited use towards achieving this goal, but for the most part the hardware is context-insensitive in this regard. Hence, there have been new architecture proposals, such as tagged architectures meant to provide context at the hardware level.

In the larger picture, hardware support is important so that interaction with the hardware layer can be done in a safe manner. However, hardware support alone cannot provide complete protection in the software layers. The operating system layer needs to be aware of the hardware support to fully make use of it and the same is true for applications. Partial support or even isolation is of course possible, but a fully memory safe system would need to be context aware across the whole stack.

4.3 Limits of Software

On the other hand, full memory safety for a system cannot be done purely in software. At best, some part of the system would abstract out unsafe communication with the hardware and serve as a middle layer for other components. In existing systems this is one of the tasks of the kernel.

However, even within the scope of individual applications, full memory safety can be incomplete. The software language itself imposes limits on how secure an application can be made. In order to provide flexibility for programmers, languages can introduce ambiguities in behavior from the system's perspective. C in particular has a great deal of undefined or implementation defined behavior that is widely used.

4.3.1 C Corner Cases

C corner cases are behaviors and features of the language that prove problematic for protection systems. The problems may be due to breaking legacy compatibility, required ambiguity due to the language specification, or both. There exist examples that complicate both spatial and temporal safety, although spatial cases have been explored much more.

4.3.2 Corner Case Classification

To better grasp the nuances of these cases, they can be classified based on two binary features: defined and securable.

Defined

The defined feature describes whether a corner case is based on defined or undefined behavior per the language specification. The distinction is useful because it highlights what cases are inherent to the language and what cases are a result of programmers relying on undefined behavior.

Securable

The securable trait refers to whether a corner case can be fixed without breaking requirements set by the language specification. The fix may involve preventing the relevant behavior entirely or building a mechanism around it to avoid its pitfalls. This characteristic is indicative of what behaviors are impossible to secure due to the constraints imposed by the language design.

4.3.3 Example Corner Cases

The following examples demonstrate the classification dimensions applied to actual corner cases:

1. *Defined and Unsecurable:* A pointer to a struct and a pointer to the first element of that struct are considered identical according to the C specification. This requirement introduces ambiguity that can complicate bounds checking. For example, a protection scheme is unable to narrow the bounds on such a pointer without breaking the required flexibility in the pointer's meaning. Thus, this behavior is defined but unsecurable since the ambiguity is a result of the language specification.

2. *Undefined and Securable:* Casting pointers to integers (downcasting) and vice versa (upcasting) is a commonly used behavior in C programs that is implementation-defined. A heavy-handed approach to secure this case which does not care for legacy compatibility is to simply ban the behavior entirely.

3. *Undefined and Unsecurable:* The order in which sequence point modifications to the values of objects occurs is not defined. In many cases the result of this behavior is benign. For example, the expression

   ```
   f() + g()
   ```

   may have either of the functions `f` or `g` evaluated first but the end result will be the same. On the other hand, when the same value is modified multiple times in the same expression such as in

   ```
   a = a++
   ```

   the result is undefined. The increment example may seem slightly contrived, but it is easy to imagine more complicated expressions that happen to modify the same value somewhere along the way (intentionally or unintentionally).

Many spatial cases were discovered and explored by the CHERI team as they were building and testing their system [11, 30]. However, those cases are all examples of undefined (or implementation-defined) and securable behaviors.

Besides spatial cases, there are also temporal issues that need to be considered. These cases can be hard to discover since they typically involve dynamic behavior. One example of a temporal corner case can be referred to as forced memory reuse. The core idea is a variation of a use after free vulnerability: a pointer is freed and later reallocated such that it points to a valid but different object when used. The forced part comes from the idea that in order to guarantee what object the pointer will point to, an attacker can fill up memory before freeing the object. Therefore, when the system handles the new memory allocation it must be to the same part of memory that was freed.

Interestingly, the forced memory reuse case will go undetected in temporal safety protection schemes that only check the validity of an object since the pointer can always point to a valid object when dereferenced. As for characterization, it can be considered a defined and securable behavior. However, the potential fixes to address this case, such as avoiding virtual address reuse, are not commonly deployed and can pose considerable overhead depending on their implementation.

4.3.4 Corner Cases Analysis

To understand whether these cases exist in real programs we conduct some static analysis on a subset of commonly used C programs and libraries.

LLVM Infrastructure

Using LLVM, many of these cases can be detected and the code can even be accordingly modified.

LLVM provides Clang as a frontend that can be used to analyze source code directly through its corresponding abstract syntax tree (ABS). It also provides a backend that can be used to analyze the intermediate code (IR) produced by the compiler. Information can be more easily accessible through one or the other and there are various interfaces to interact with the available data.

Example Passes and Plugins

As an example, for the struct pointer ambiguity case, the Clang frontend can be used to find definitions of structs that have an array as their first member. Figure 4-1 shows a snippet of a Clang plugin that accomplishes this task.

Furthermore, the number of uses of such a struct in a program can be obtained by with a pass using the LLVM backend. Such a pass can check `GetElementPointer` instructions to check if they are accessing a struct with a buffer as its first member. `GetElementPointer` instructions are produced in LLVM IR whenever a pointer is dereferenced. A snippet of the code that accomplishes this is shown in Figure 4-2.

Both of these examples simply output metadata data along with a message whenever an instance is found. However, the same plugin and pass can be easily modified to change the code or IR when it finds such a case. This ability can be useful for various scenarios such as automatically implementing a fix if possible and/or specified

```
virtual void HandleTagDeclDefinition(TagDecl *D) {
  SourceManager &sourceManager = Instance.getSourceManager();
  if (D->isStruct()) {
    const RecordDecl *r = dyn_cast<RecordDecl>(D);
    RecordDecl::field_iterator firstMember = r->field_begin();
    if (firstMember->getType()->isArrayType()) {
      llvm::errs()
        << D->getLocation().printToString(sourceManager) << " "
        << "Struct:" << r->getNameAsString() << " "
        << "Array:" << firstMember->getNameAsString()
        << ’\n’;
    }
  }
}
```

Figure 4-1: A code snippet used for detecting definitions of structs with buffers as their first member.

through a compiler flag.

Sample Results

Table 4.1 shows the number of uses found for a few of the corner cases by using the LLVM infrastructure. As can be seen most of these cases occur with non-negligible frequency.

The CHERI team also provided their own analysis of the existence and usage of the cases they found [11].

Completeness

Some of the cases can be more complicated to find and analyze and others are better performed through dynamic analysis (e.g. temporal corner cases). Such an analysis would be useful in creating a tool to help annotate programs or otherwise fix issues resulting from these behaviors, but it is unclear if such a solution could provide completeness.

Moreover, there is the issue that detection methods or tools are likely to have many false positives or false negatives for more complicated cases that required a more complicated analysis.

```
virtual bool runOnBasicBlock(BasicBlock &BB) {
  for (BasicBlock::iterator ii = BB.begin(), ii_e = BB.end();
      ii != ii_e; ++ii) {
    if (GetElementPtrInst *gep = dyn_cast<GetElementPtrInst>(&*ii)) {
      Type *srcElem = gep->getSourceElementType();
      if (StructType *srcStruct = dyn_cast<StructType>(srcElem)) {
        if (isStructWithBufferFirst(srcStruct)) {
          printDebugInfo(&*ii);
          errs() << srcStruct->getName().str() << ": ";
          if (gep->getResultElementType()->isArrayTy()) {
            errs() <<
              "Found buffer-first struct access to array member." << ’\n’;
          } else {
            errs() <<
              "Found buffer-first struct access to other member." << ’\n’;
          }
        }
      }
    }
  }

  return false;
}
```

Figure 4-2: A code snippet used for detecting accesses into structs with buffers as their first member.

Other Analyses

Besides collecting usage statistics for the corner cases, other analyses may provide some insights into how problematic they are. For example, an exploitability analysis would prove interesting in seeing how many of these cases result in immediately viable exploits. However, determining whether a particular case is exploitable could prove complicated especially for some of the more general corner cases. In addition, there is the inherent issue that if a case is left unaltered because it was considered to be benign at one point, that does not prevent it from becoming exploitable due to later changes in the program.

	Pointer Subtraction	Buffer First Struct	Upcasting / Downcasting	Pointer Shrinkage	Relative Pointer Comparison
git	586	58	675	0	248
httpd	7	20	3	0	0
lighttpd	117	15	162	0	18
nginx	454	18	613	0	132
openssl	189	48	274	0	71
postgresql	570	60	3869	0	555
redis	112	22	111	0	46
acl	19	5	42	0	3
bash	132	12	154	0	74
coreutils	294	33	261	0	171
e2fsprogs	61	22	167	0	24
findutils	77	16	62	0	35
gcc	165	34	207	0	79
grep	0	3	0	0	0
gzip	13	11	12	0	5
ncurses	111	15	112	0	50
pam	0	1	0	0	0
sed	0	0	0	0	0
slang	308	21	167	0	370
tar	0	0	0	0	0
zlib	37	4	16	0	9

Table 4.1: Number of occurrences of a selection of corner cases in commonly used C programs and the most used programs in Ubuntu's popularity contest.

4.3.5 Benefits of Classification

Classifying corner cases based on whether they are defined and securable presents a clearer picture of the limits of safety in the language. On the one hand, the defined characteristic illustrates the behaviors that may be used but cannot be relied on. On the other, the securable characteristic shows the line between what a defense system can hope to accomplish in the best possible case and what is out of scope due to the language specification.

4.4 Scoping the Goal

Thus, achieving full memory safety requires cooperation across all levels of the stack from hardware through the application software layer. Nevertheless, the transition is unlikely to happen all at once, so defenses have focused on providing protections at different layers. Unfortunately, the goal of complete memory safety is at odds with backwards compatibility. From context-insensitive hardware to language ambiguities, the amount of protection that can be afforded to existing systems is restricted.

New designs need to be aware of the limits imposed by the environment they choose to function in, whether that is in hardware or software. This awareness is critical to accurately determining what level of protection can be afforded even for new systems designed to replace the existing flawed ones.

Language limits to memory safety also need to be explored in the case that a more secure language is chosen to create a more secure system. It would be interesting to further explore how many corner cases still exist in variants of C that focus on providing greater secure such as Cyclone [25] and CCured [36]. However, even beyond C it's possible for language specifications to introduce other limitations or corner cases that can complicate securing a system.

Chapter 5

Conclusion

Memory safety is a topic that has been discussed and analyzed at length. Many designs have been proposed to provide memory safety guarantees through hardware, software, or a combination of both. However, most of these designs have attempted to provide only some partial degree of safety to minimize overhead. For example, there have been numerous variations of allocation bounds schemes that attempt to keep overhead low. The problem is that these schemes only minimally increase the difficulty of mounting an attack rather than providing a more complete fix.

Full memory safety is needed in order to avoid all the pitfalls that partial memory safety systems have. Given new designs and performance improvements, it is becoming more feasible to imagine a system that can provide full memory safety. However, while providing full memory safety sounds like a good overall goal, the specifics have many details that need to be carefully considered. While partial protection systems have clear requirements based on what they intend to protect, the requirements needed to obtain full memory safety are less clearly defined.

Moreover, full memory safety remains at odds with backwards compatibility. Existing systems impose limits to what protection can actually be provided. A major example of this is the C language. It was purposefully designed to be extremely flexible, but a byproduct of that flexibility is corner cases that prevent protection schemes from providing complete protection. Thus, any systems that intend to provide full memory safety will have inherent limitations if they intend to function on a legacy software stack.

Ultimately, any system that intends to provide truly complete memory safety will likely have to break compatibility with most existing hardware and software stacks. At best, to preserve compatibility a protection system may isolate components that do not adhere to its full memory safety requirements. However, eventually those components would need to be replaced or rewritten in order to provide a fully memory safe system.

Bibliography

[1] Cve-2009-2629, 2009.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Stevenb Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[3] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on very large scale integration (VLSI) systems*, 15(5):546–559, 2007.

[4] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *ACM SIGPLAN Notices*, 49(1):165–178, 2014.

[5] Hans Benker, Jean-Michel Beacco, M Dorochevsky, Th Jeffré, A Pöhlmann, J Noye, B Poterie, JC Syre, O Thibault, and G Watzlawik. Kcm: a knowledge crunching machine. *ACM SIGARCH Computer Architecture News*, 17(3):186–194, 1989.

[6] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[7] Jeremy Brown and Thomas F Knight Jr. A minimal trusted computing base for dynamically ensuring secure information flow. *Project Aries TM-015 (November 2001)*, 37, 2001.

[8] Nicholas P Carter, Stephen W Keckler, and William J Dally. Hardware support for fast capability-based addressing. In *ACM SIGPLAN Notices*, volume 29, pages 319–327. ACM, 1994.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[10] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 14–27. ACM, 2012.

[11] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *ACM SIGPLAN Notices*, volume 50, pages 117–130. ACM, 2015.

[12] Jedidiah R Crandall, S Felix Wu, and Frederic T Chong. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):359–389, 2006.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 482–493. ACM, 2007.

[14] Daniel Y Deng and G Edward Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[15] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 103–114. ACM, 2008.

[16] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *ACM SIGARCH Computer Architecture News*, 43(1):487–502, 2015.

[17] Udit Dhawan, Albert Kwon, Edin Kadric, Catalin Hritcu, Benjamin C Pierce, Jonathan M Smith, André DeHon, Gregory Malecha, Greg Morrisett, Thomas F Knight, et al. Hardware support for safety interlocks and introspection. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*, pages 1–8. IEEE, 2012.

[18] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 8. ACM, 2014.

[19] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[20] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM.

[21] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, Feb 2017.

[22] Samuel Fingeret. *Defeating Code Reuse Attacks with Minimal Tagged Architecture*. PhD thesis, MIT CSAIL, 2015.

[23] Julián Armando González. *Taxi: Defeating Code Reuse Attacks with Tagged Memory*. PhD thesis, Massachusetts Institute of Technology, 2015.

[24] intel. Introduction to intel memory protection extensions, 2013.

[25] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[26] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[27] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 105–114. IEEE, 2009.

[28] Albert Kwon, Udit Dhawan, Jonathan Smith, Thomas Knight, and Andre Dehon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732. ACM, 2013.

[29] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[30] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of c: elaborating the de facto standards. In *ACM SIGPLAN Notices*, volume 51, pages 1–15. ACM, 2016.

[31] David A Moon. Architecture of the symbolics 3600. *ACM SIGARCH Computer Architecture News*, 13(3):76–83, 1985.

[32] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Hardware-enforced comprehensive memory safety. *IEEE Micro*, 33(3):38–47, 2013.

[33] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 175. ACM, 2014.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[36] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*, 37(1):128–139, 2002.

[37] Elliott I Organick. *Computer system organization: the B5700/B6700 series*. Academic Press, 1973.

[38] Elliott I Organick. *Computer system organization: the B5700/B6700 series*. Academic Press, 1973.

[39] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. Sift: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, page 37. ACM, 2011.

[40] Stephen Phillips. M7: Next generation sparc. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–27. IEEE, 2014.

[41] Krerk Piromsopa and Richard J Enbody. Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing*, 3(4):365–376, 2006.

[42] Joël Porquet and Simha Sethumadhavan. Whisk: An uncore architecture for dynamic information flow tracking in heterogeneous embedded socs. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 4. IEEE Press, 2013.

[43] Raj Prakash. The holy grail - real time memory access checking, 2015.

[44] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Christopher Liebchen Stephen Crane, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, Feb 2017.

[45] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[46] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th*, S&P, 2015.

[47] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[48] Ryota Shioya, KIM Daewung, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. Low-overhead architecture for security tag. *IEICE transactions on information and systems*, 94(1):69–78, 2011.

[49] Howard Shrobe, Andre DeHon, and Thomas Knight. Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (tiara). Technical report, DTIC Document, 2009.

[50] Burton J Smith. Architecture and applications of the hep multiprocessor computer system. In *25th Annual Technical Symposium*, pages 241–248. International Society for Optics and Photonics, 1982.

[51] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. 2016.

[52] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[53] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices*, volume 39, pages 85–96. ACM, 2004.

[54] Gregory T Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The dover inherently secure processor. In *Technologies for Homeland Security (HST), 2017 IEEE International Symposium on*, pages 1–5. IEEE, 2017.

[55] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[56] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I

August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 243–254. IEEE, 2004.

[57] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184. IEEE, 2008.

[58] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016.

[59] Robert NM Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, Munraj Vadera, and Khilan Gudka. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015.

[60] Emmett Witchel, Josh Cates, and Krste Asanović. *Mondrian memory protection*, volume 30. ACM, 2002.

[61] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[62] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. ACM.

[63] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, volume 8, pages 225–240, 2008.

[64] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. Spur lisp: Design and implementation. Technical report, Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, 1987.