The Continuing Arms Race

Code-Reuse Attacks and Defenses

Edited by
Per Larsen
Ahmad-Reza Sadeghi

acm BOOKS
M&C

The Continuing Arms Race

ACM Books

Editor in Chief

M. Tamer Özsu, *University of Waterloo*

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

The Continuing Arms Race: Code-Reuse Attacks and Defenses

Editors: Per Larsen, *Immunant, Inc.*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*
2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun, *University of California, Berkeley*
2017

Computational Prediction of Protein Complexes from Protein Interaction Networks

Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*
Chern Han Yong, *Duke-National University of Singapore Medical School*
Limsoon Wong, *National University of Singapore*
2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations

Editors: Sharon Oviatt, *Incaa Designs*
Björn Schuller, *University of Passau and Imperial College London*
Philip R. Cohen, *Voicebox Technologies*
Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*
Gerasimos Potamianos, *University of Thessaly*
Antonio Krüger, *German Research Center for Artificial Intelligence (DFKI)*
2017

Communities of Computing: Computer Science and Society in the ACM

Thomas J. Misa, Editor, *University of Minnesota*
2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining
ChengXiang Zhai, *University of Illinois at Urbana–Champaign*
Sean Massung, *University of Illinois at Urbana–Champaign*
2016

An Architecture for Fast and General Data Processing on Large Clusters
Matei Zaharia, *Stanford University*
2016

Reactive Internet Programming: State Chart XML in Action
Franck Barbier, *University of Pau, France*
2016

Verified Functional Programming in Agda
Aaron Stump, *The University of Iowa*
2016

The VR Book: Human-Centered Design for Virtual Reality
Jason Jerald, *NextGen Interactions*
2016

Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age
Robin Hammerman, *Stevens Institute of Technology*
Andrew L. Russell, *Stevens Institute of Technology*
2016

Edmund Berkeley and the Social Responsibility of Computer Professionals
Bernadette Longo, *New Jersey Institute of Technology*
2015

Candidate Multilinear Maps
Sanjam Garg, *University of California, Berkeley*
2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing
John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*
2015

A Framework for Scientific Discovery through Video Games
Seth Cooper, *University of Washington*
2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers
Bryan Jeffrey Parno, *Microsoft Research*
2014

Embracing Interference in Wireless Systems
Shyamnath Gollakota, *University of Washington*
2014

The Continuing Arms Race

Code-Reuse Attacks and Defenses

Per Larsen
Immunant, Inc.

Ahmad-Reza Sadeghi
Technische Universität Darmstadt

ACM Books #18

The Continuing Arms Race

Per Larsen, Ahmad-Reza Sadeghi, editors

books.acm.org
www.morganclaypoolpublishers.com

ISBN: 978-1-97000-183-9 hardcover
ISBN: 978-1-97000-180-8 paperback
ISBN: 978-1-97000-181-5 eBook
ISBN: 978-1-97000-182-2 ePub

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3129743 Book
10.1145/3129743.3129744 Preface
10.1145/3129743.3129745 Chapter 1
10.1145/3129743.3129746 Chapter 2
10.1145/3129743.3129747 Chapter 3
10.1145/3129743.3129748 Chapter 4
10.1145/3129743.3129749 Chapter 5
10.1145/3129743.3129750 Chapter 6
10.1145/3129743.3129751 Chapter 7
10.1145/3129743.3129752 Chapter 8
10.1145/3129743.3129753 References

A publication in the ACM Books series, #18
Editor in Chief: M. Tamer Özsu, *University of Waterloo*

First Edition

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Our societies are becoming increasingly dependent on emerging technologies and connected computer systems that are increasingly trusted to store, process, and transmit sensitive data. While generally beneficial, this shift also raises many security and privacy challenges. The growing complexity and connectivity offers adversaries a large attack surface. In particular, the connection to the Internet facilitates remote attacks without the need for physical access to the targeted computing platforms. Attackers exploit security vulnerabilities in modern software with the ultimate goal of taking control over the underlying computing platforms. There are various causes of these vulnerabilities, the foremost being that the majority of software (including operating systems) is written in unsafe programming languages (mainly C and C++) and by developers who are by-and-large not security experts.

Memory errors are a prominent vulnerability class in modern software: they persist for decades and still are used as the entry point for today's state-of-the-art attacks. The canonical example of a memory error is the stack-based buffer overflow vulnerability, where the adversary overflows a local buffer on the stack, and overwrites a function's return address. While modern defenses protect against this attack strategy, many other avenues for exploitation exist, including those that leverage heap, format string, or integer overflow vulnerabilities.

Given a memory vulnerability in the program, the adversary typically provides a malicious input that exploits this vulnerability to trigger malicious program actions not intended by the benign program. This class of exploits aims to hijack the control flow of the program and differs from conventional malware, which encapsulates the malicious code inside a dedicated executable that needs to be executed on the target system and typically requires no exploitation of a program bug.

As mentioned above, the continued success of these attacks is mainly attributed to the fact that large portions of software programs are implemented in type-unsafe languages (C, C++, or Objective-C) that do not guard against malicious program inputs using bounds checking, automatic memory management, etc. However, even

type-safe languages like Java rely on virtual machines and complex runtimes that are in turn implemented in type-unsafe languages out of performance concerns. Unfortunately, as modern applications grow more complex, memory errors and vulnerabilities will likely continue to exist, with no end in sight.

Regardless of the attacker's method of choice, exploiting a vulnerability and gaining control over an application's control flow is only the first step of an attack. The second step is to change the behavior of the compromised application to perform malicious actions. Traditionally, this has been realized by injecting malicious code into the application's address space, and later executing the injected code. However, with the widespread enforcement of data execution prevention (DEP), such attacks are more difficult to launch today. Unfortunately, the long-held assumption that only code injection posed a risk was shattered with the introduction of code-reuse attacks, such as return-into-libc and return-oriented programming (ROP). As the name implies, code-reuse attacks do not require any code injection and instead repurpose benign code already resident in memory.

Code-reuse techniques are applicable to a wide range of computing platforms: x86-based platforms, embedded systems running on an Atmel AVR processor, mobile devices based on ARM, PowerPC-based Cisco routers, and voting machines deploying a z80 processor. Moreover, the powerful ROP technique is Turing-complete, i.e., it allows an attacker to execute arbitrary malicious code.

In fact, the majority of state-of-the-art run-time exploits leverage code-reuse attack techniques, e.g., against Internet Explorer, Apple QuickTime, Adobe Reader, Microsoft Word, or the GnuTLS library. Even large-scale cyberattacks such as the popular Stuxnet worm, which damaged Iranian centrifuge rotors, incorporated code-reuse attack techniques.

Indeed, even after more than three decades, memory corruption exploits remain a clear and present danger to the security of modern software and hardware platforms. This is why the research community both in academia and industry have invested major efforts in the recent past to mitigate the threat. Various defenses have been proposed and even deployed by Google, Microsoft, Intel, etc. The most prominent defenses are based on enforcement (e.g., Control-Flow Integrity [CFI]) or randomization (also known as software diversity) of certain program aspects. Both types of defenses have distinct advantages and disadvantages. Randomization makes it hard for attackers to chain together their attack gadgets, is efficient, and can be applied to complex software such as web browsers. It can have different levels of granularity, from a simple Address Space Layout Randomization (ASLR) to fine-grained randomization at function or even instruction level. However, randomization requires high entropy and all randomization schemes are inherently

vulnerable to information leakage. CFI, on the other hand, provides guarantees that the application does not deviate from the intended program flow, yet it requires a security and efficiency tradeoff: i.e., coarse-grained CFI have been shown to be vulnerable, and fine-grained CFI can be inefficient without hardware support. Researchers have been working on improving these schemes with novel ideas in both software and hardware design. Many proposed defenses have been bypassed by other attacks, generating a large body of literature on this topic.

It seems that the arms race between attackers and defenders continues. Although researchers have raised the bar for adversaries, there are still a number of challenges to tackle against sophisticated attacks. Despite all the recent proposals on various defenses, we cannot claim that the problem is entirely solved. However, our community has gained much insight through recent results on how and to what extent we need to employ certain design principles to significantly reduce the effect of code-reuse attacks.

The main purpose of this book is to provide readers with some of the most influential works on run-time exploits and defenses. We hope that this material will inspire readers and generate new ideas and paradigms.

Per Larsen
Ahmad-Reza Sadeghi
February 2018

1 How Memory Safety Violations Enable Exploitation of Programs

Mathias Payer

Our programs, runtime systems, operating systems, and hypervisors are, to a large extent, written in low-level languages like C or C++. These systems languages were initially designed more than 30 years ago when performance was the key metric and security was, at best, a side note. Our systems languages do not enforce memory safety and force the programmer to include necessary safety checks. Coding guidelines and code quality continuously improved over time due to an increased awareness of security. Along with this awareness, language standards and compilers evolved as well and became more powerful, especially compilers that now offer (optional) safety checks. Yet, despite these improvements some attack surface remains. Adversaries can abuse bugs that cause memory safety violations to change the semantics of the program, executing the adversary's desired behavior. Memory safety issues are a problem not just for legacy software that was developed decades ago but also for new software, such as Google Chrome or Microsoft Edge, both large software projects that started just a couple of years ago under strict coding and testing guidelines.

Missing checks in source code are the root cause of different forms of memory safety issues. Examples of such safety issues are (i) buffer overflows or arbitrary memory corruption if a pointer dereferences memory outside the bounds of the underlying memory object, (ii) use-after-free conditions if a reference to a previously freed memory object is dereferenced, and (iii) type confusion if a base object is cast down to a subtype. Invalid pointers are easily created by mistake: sloppy pointer arithmetic, off-by-N mistakes in loop bounds, or integer errors (e.g., an

integer value is incremented past its maximum and flips into a negative value or a truncated type is used in a comparison, especially when used with memory allocation functions). Adversaries can leverage these bugs to change the semantics of the executing program by, e.g., modifying variables that are used in comparisons outside of regular control flow, overwriting code pointers (references in the data region that point to a code segment) to redirect execution, or leaking information by copying sensitive data to an output stream.

There is a fundamental disparity in difficulty between defending software and attacking it. While a programmer must fix all possible bugs and defenses must cover all possible attack vectors to secure a system, an adversary only needs to find one exploitable bug to gain control of the system. This disparity makes defense challenging. Finding and fixing through formal methods, static bug-finding techniques, or fuzz testing is important. However, due to the large state space of programs, it is unlikely that all bugs will ever be found. Systems therefore must rely on defense mechanisms to protect themselves against attacks. Each defense mechanism enforces some security policy. Runtime monitors observe the state of the program according to the underlying security policy and terminate the process if a violation is detected. When designing defenses, researchers must consider if the defense will fundamentally stop an attack vector and provide a real solution or if it will merely "raise the bar." Arguing how much a defense raises the bar is challenging and may be questionable. For example, adversaries tend to invent and develop automated tools that bypass defenses. A fundamental problem when evaluating defenses is the lack of quantitative metrics and benchmarks to assess security qualities and measuring *how much* and *how effectively* a defense raises the bar. Another interesting design decision is *where* or *when* the attack is detected. Memory safety stops the process before the adversary writes a single illegal byte to the address space while other mechanisms such as stack canaries allow modification of the memory but detect corruption later, when the data is used. Security is an arms race between defenders slowly improving defenses while ensuring availability and compatibility while adversaries continue to probe for new weaknesses.

We discuss the attack surface through memory safety errors for applications and runtime systems written in low level languages. In the *Eternal War in Memory* survey [Szekeres et al. 2013, Szekeres et al. 2014], the authors discuss defenses along four attack vectors: code injection, control-flow hijacking, data-only attacks, and information leaks. Attacks follow the path of least resistance, subject to the adversary's constraints. From a high-level perspective, an adversary wants to (i) inject, change, or modify code, (ii) hijack the control flow of the application to

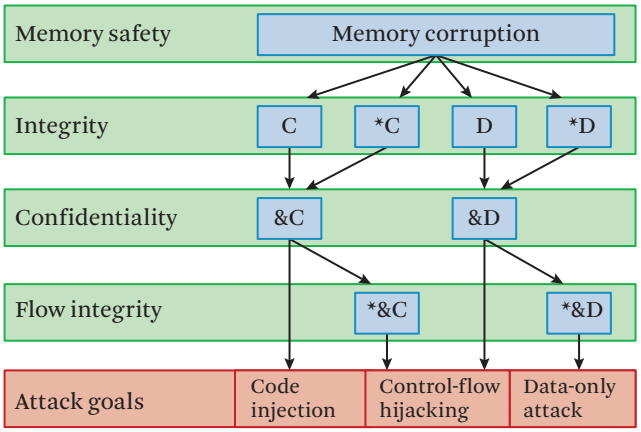

Figure 1.1 Attack surface, showing attack paths from the initial memory safety violation to code injection, control-flow hijacking, and data-only attacks. In the figure, C shows code, D shows data, asterisk (*) marks pointers, ampersand (&) marks addresses, and *& marks a dereference operation, e.g., *&C tells us that a code pointer is being dereferenced.

locations that are outside the valid control flow, or (iii) modify data in some way to either leak information or set up non-control-data attacks.

Figure 1.1 shows the attack surface reachable through memory safety errors. An adversary may leverage memory safety errors to execute any of the pictured attack paths. Defenses can stop the attack at any layer before the attack successfully completes. Note that defenses at any layer always allow the adversary to launch a denial of service attack as the underlying bug is still in the program and defenses are (usually) restricted to terminating the offending thread or process. Recovery from memory safety violations without program-specific recovery code is hard due to the semantic gap between execution, source code, and programmer intention. The root cause of the violation could be manifold, e.g., an invalid pointer computation, a memory object that was freed too early, or a mishap in an integer computation. Defenses therefore opt to terminate the offending thread or process, potentially causing denial of service but protecting the integrity of the system and the confidentiality of data.

Starting from a defender's perspective, we follow Figure 1.1 and discuss different security policies, beginning with memory safety at a high level in Section 1.1, followed by data integrity in Section 1.2, confidentiality in Section 1.3, and data-flow integrity in Section 1.4. Security policies running on a system can be enforced

through different mechanisms, which are introduced in Section 1.5. Moving to the adversary's point of view, we introduce techniques and practices that achieve the desired level of control in Section 1.6. These techniques allow understanding of fundamental attack vectors used to compromise systems through memory safety vulnerabilities. We then finish with concluding remarks in Section 1.7.

1.1 Memory Safety

Memory corruption, the absence of memory safety, is the root cause of current high-profile attacks and the foundation of a plethora of different attack vectors. Memory safety is a program property that guarantees memory objects can only be accessed with the corresponding capabilities. At an abstract level, a pointer is a capability to access a certain memory object or memory region [Hicks 2014, Nagarakatte et al. 2009]. A pointer receives capabilities whenever it is assigned and is then allowed to access the pointed-to memory object. The capabilities of a memory object describe the size or area, validity, and type of the underlying object. Capabilities are assigned to a memory object when it is created. The initial pointer returned from the memory allocator receives these capabilities and can then pass them, through assignment, to other pointers. Memory objects can be created explicitly by calling the allocator, implicitly for global data by starting the program, or implicitly for the creation of a stack frame by calling a function. The capabilities are valid as long as that memory object remains alive. Pointers that are created from this initial pointer receive the same capability and may only access the object inside the bounds of that object, and only as long as that object has not been deallocated. Deallocation, either through an explicit call to the memory allocator or through removal of the stack frame by returning to the caller, destroys the memory object and invalidates all capabilities.

Pointer capabilities cover three areas: bounds, validity, and type. The bounds of a memory object encode spatial information of the memory object. *Spatial memory safety* ensures that pointer dereferences are restricted to data *inside* the memory object. Memory objects are only valid as long as they are allocated. *Temporal safety* ensures that a pointer can only be dereferenced as long as the underlying object *stays allocated*. Memory objects can only be accessed if the pointer has the correct type. *Type safety* ensures that the object's type matches one of the *compatible types* according to type inheritance. The C/C++ family of programming languages allows invalid pointers to exist, i.e., a pointer may point to an invalid memory region that is out of bounds or no longer valid. A memory safety violation only triggers when such an invalid pointer is dereferenced.

Memory safety can be enforced at different layers. Language-based memory safety makes it impossible for the programmer to violate memory safety by, e.g., checking each memory access and type cast (Java, C#, or Python) or enforcing a strict static type system (Rust). Systems that retrofit memory safety to C/C++ are commonly implemented at the compiler level due to the availability of pointer and type information. Techniques that retrofit memory safety for C/C++ must track each pointer and its associated bounds for spatial memory safety, validity for temporal memory safety, and associated type for type safety.

1.1.1 Spatial Memory Safety

Spatial memory safety ensures that a pointer can only dereference data that is within the bounds of the assigned object. If an out-of-bounds pointer is dereferenced, a spatial memory safety violation is signaled, terminating the process. Language-based solutions like CCured [Necula et al. 2005] and Cylcone [Jim et al. 2002] enforce a stricter type system on top of the loose typing that C/C++ offer and make memory safety constraints explicit. Compiler-based solutions like SoftBound [Nagarakatte et al. 2009] allocate a disjoint metadata store to track all pointers and their associated bounds. In addition, both language-based and compiler-based solutions instrument pointer assignment and pointer arithmetic to propagate the underlying bounds information and prepend every dereference with an explicit check of whether the pointer is still valid. SoftBound favors a disjoint metadata storage where the address of the pointer is used to look up metadata information. Disjoint metadata eases portability and enables seamless integration of memory safety solutions [Nagarakatte et al. 2015].

Spatial memory safety violations happen if a pointer is incremented past the bounds of the object, e.g., in a loop or through pointer arithmetic:

```
char *c = (char*)malloc(16);
for (int i = 0; i <= 16; i++) {
  // buffer overflow for i == 16
  *c = i;
}
// violation through a direct write
c[16] = 42;
```

1.1.2 Temporal Memory Safety

Temporal memory safety ensures that the pointer can only reference live memory objects. If the underlying memory object is no longer valid, e.g., because it is freed (for heap objects) or the function returned (for stack objects), dereferencing a

stale pointer results in undefined behavior. CETS [Nagarakatte et al. 2010] retrofits temporal memory safety on top of C/C++. In addition to metadata for each pointer, CETS also allocates metadata for each object. Both pointer and object receive an associated identifier, and only if the identifiers match is the object valid and the dereference operation allowed.

Temporal memory safety violations happen if the underlying memory object was freed:

```
char *c = malloc(16);
char *d = c;
free(d);
// violation as c no longer points to a valid object
c[12] = 23;
```

1.1.3 Type Safety

Type safety is a programming language concept that assigns each allocated memory object an associated type. Typed memory objects may only be used at program locations that expect the corresponding type. Casting operations allow an object to be interpreted under a different type. Casting is allowed along the inheritance chain. Upward casts (upcasts) move the type closer to that of the root object so that the type becomes more generic, while downward casts (downcasts) specialize the object to a subtype. For example, consider an inheritance chain from the root object Animal over Mammal to the specialized type Ape. If we have a reference of type Mammal, an upcast could cast a Mammal into an Animal and a downcast could cast a Mammal into an Ape. The type hierarchy is specified by the programmer according to the semantics of the source programming language. In low-level languages like C or C++, type safety is not explicit and a memory object can be reinterpreted in arbitrary ways. C++ provides a vast set of type cast operations. Static casts are only checked at compile time to ensure that the two types are compatible. Dynamic casts execute a slow runtime check, which is only possible for polymorphic classes with virtual functions; otherwise, no vtable pointer—to identify the object's type—is available in the memory object layout. Reinterpret casts allow reclassification of a memory object under a different type. Static casts have the advantage that they do not incur any runtime overhead but are purely checked at compile time. Static casts lack any runtime guarantees, and objects of the wrong type may be used at runtime. For example, Listing 1.1 shows a type violation where an object of the base type can be used as a subtype after an illegal downcast. Reinterpretation of casts allows the programmer to explic-

```
class B {
  int b;
};
class D: B {
  int c;
  virtual void d() {}
};
.
.
.
B *Bptr = new B;
// Type violation:
D *Dptr = static_cast<D*>B;
Dptr->c = 0x43; // Type confusion!
Dptr->d();      // Type confusion!
```

Listing 1.1 Type violation after illegal downcast.

itly break the underlying type assumptions and reassign a different type to the pointer or underlying memory object. Due to the low-level nature of C++, a programmer may write to the raw memory object and change the underlying object directly.

Ideally, a program can statically be proven type safe. Unfortunately, this is not possible for C/C++ and defenses have to resort to runtime checks. If we make all casts in the program explicit and check them for correctness, then we can ensure that the runtime type conforms to the statically assumed type at compile time. UBSan [Project 2013] follows this approach. Unfortunately, enforcing dynamic casts results in prohibitive performance overhead and compatibility issues. UBSan requires manual blacklisting for non-polymorphic classes. In addition, the existing type-checking infrastructure in C++ was optimized under the assumption that only few runtime checks would be executed and is therefore inherently slow. CaVer [Lee et al. 2015] vastly extends the coverage of UBSan by using a disjoint metadata table (similar to other memory safety techniques) to allow explicit type checks on non-polymorphic objects as well. Unfortunately, the instrumentation results in prohibitively high performance overhead. Also, CaVer only supports objects on the heap allocated through the `new` operator; stack objects (`alloca`) and other heap objects (`malloc`) are not supported. TypeSan [Haller et al. 2016] extends coverage over CaVer and reduces performance overhead, enabling an always-on type-checking solution.

1.2 Data Integrity

Data integrity is a policy that ensures integrity of data in the process's address space. For efficiency reasons, individual data integrity policies only protect a subset of data. The protected data may depend on the underling type, location, or when it was allocated. Defense mechanisms often rely on integrity as a basis, i.e., protecting code regions or data regions against modification. For example, any defense that relies on some form of runtime check relies on code integrity as the adversary could otherwise simply rewrite the underlying code, removing the checks. Data integrity is often implemented through hardware extension, e.g., virtual memory allows setting data on a per-page basis as readable, writable, and/or executable. Each virtual memory page may have different permissions, and the memory management unit issues a page fault if the permissions do not match, e.g., a read-only page is written, an unmapped page is read, or control is transferred to a non-executable page. Software techniques can increase the flexibility and granularity but usually result in higher overhead. Write-Integrity Testing (WIT) [Akritidis et al. 2008] is a software integrity approach. WIT uses points-to analysis at compile time to assign a label to each object and to each memory write. At runtime, the label of each object is recorded and write instructions verify that the labels match. The effectiveness of WIT is limited by the completeness and over-approximation of the points-to analysis when individual labels are merged due to imprecision. Software-based Fault Isolation (SFI) [Wahbe et al. 1993, Hiser et al. 2006, Kiriansky et al. 2002, Ford and Cox 2008, Payer and Gross 2011, Payer 2012, Yee et al. 2009] allows a fine-grained separation and protection of code and data through instrumentation (and verification) of the code. Software masking, a form of SFI, can protect a certain memory area from adversary access by masking each pointer before it is dereferenced. Such a masking scheme can be implemented by prepending each dereference with a logical AND instruction of the memory address.

Most systems enforce data integrity for code and a subset of data. Code is generally marked executable and immutable. For data, the subset of immutable data, such as `const` objects, is generally marked read-only, as are support data structures used by the dynamic loader, such as symbol tables, linkage tables between different libraries, or the kernel export table.

1.2.1 Code Integrity

Code integrity enforces that existing code in a process is immutable. Combining non-executable data (e.g., through virtual memory) and code integrity results in $W \oplus X$—writable xor executable data [de Raadt 2005, PaX Team 2004b]. Data Execu-

tion Protection (DEP) [Andersen and Abella 2004] can be implemented in two ways: through non-executable data or by instruction set randomization. Non-executable data ensures that only well-marked regions that contain code can be executed. Before the introduction of the no-execute flag on commodity hardware by AMD and Intel around 2006, protection against code injection relied on a combination of segmentation and software techniques [van de Ven 2004], similar to software-based fault isolation [Wahbe et al. 1993, Hiser et al. 2006, Kiriansky et al. 2002, Ford and Cox 2008, Payer and Gross 2011]. Code integrity works well for static programs but is inherently incompatible with just-in-time compilation, dynamic library loading, and binary translation. These features require changing code at runtime, making the enforcement of code integrity policies challenging. For example, an adversary may have a small window of opportunity to compromise code while it is writable. Also, protection of just-in-time compiled code remains an open problem, with some defenses trying to contain control flow to the area where code is dynamically generated [Niu and Tan 2014b].

1.2.2 Code-Pointer Integrity

Ideally, full memory safety would stop all forms of memory corruption, but it is currently prohibitively expensive. Code-Pointer Integrity (CPI) [Kuznetsov et al. 2014a] is an integrity policy that enforces full safety for a subset of data (code pointers). CPI separates control data structures from non-control data structures and enforces that only program code that is supposed to change control data structures is allowed to do so. Code that handles data is not allowed to change code pointers, i.e., a data pointer is forbidden to modify a code pointer. The CPI mechanism uses a type-based analysis to identify sensitive pointers that must be protected. The compiler analysis classifies each pointer as a sensitive pointer or as a data pointer. Sensitive pointers are code pointers and the transitive closure of any pointer pointing to sensitive pointers. The set of sensitive pointers is then protected through memory safety checks, carving out a subset of protected data that includes all code pointers and any data that may be used to modify code pointers. Protecting only a subset of all data allows CPI to drastically reduce the overhead while protecting against any control-flow hijack attack or modification of a code pointer through a memory safety violation. Note that CPI also relies on code integrity, and the mechanism works well for code pointers on the heap or in global data.

1.2.3 Stack Integrity

A thread's stack is a data structure where buffers and code pointers are frequently allocated in close proximity to each other. Protecting code pointers against targeted

overwrites and buffer overflows is challenging due to the volatility and frequent changes. Stack-based protections also face the challenge of frequent allocations and deallocations of stack frames; the individual overhead must therefore be minimal. *Stack integrity* ensures that data on the stack remains integrity protected against illegal modifications.

Strong protections for function returns enforce stack integrity by leveraging the relationship between function calls and returns. A mechanism that enforces stack integrity ensures that any backward-edge transfers can only return to the most recent prior caller. This property can be enforced by storing the prior call sites in a shadow stack or guaranteeing memory safety on the stack, i.e., if the return instruction pointers are immutable, then stack integrity trivially holds. Control-flow enforcement for function returns is "easier" than for indirect function calls because function calls and returns form a symbiotic relationship that can be leveraged in the design of the defense, i.e., a function return always returns to the location of the previous call. Such a relationship does not exist for indirect function calls. Depending on the mechanism, at least return pointers are protected by shadow stacks [Payer et al. 2015c, Dang et al. 2015] and at best the majority of data for safe stacks [Kuznetsov et al. 2014a].

1.3 Confidentiality

Defenses based on confidentiality are probabilistic: if a data value is unknown, the adversary is restricted to guessing the correct value. For defenses that hide memory addresses, randomization results in a search space of $\frac{1}{2^{32}}$ for 32-bit systems or $\frac{1}{2^{64}}$ for 64-bit systems and is highly unlikely to succeed. While practical implementations cannot leverage the full virtual memory space, it remains large enough in practice. Randomization-based defenses hide locations of sensitive data, e.g., code pointers, flags, or privileged data, from an adversary. Through the use of virtual memory, data can be spaced out in a vast address space where the majority of pages remain unmapped. Without knowing the precise address, an adversary will likely trigger a segmentation fault when trying to guess the correct location. All modern systems use Address Space Layout Randomization (ASLR) [Durden 2002, Bhatkar et al. 2003, Bhatkar et al. 2005] to randomize and hide the location of the base addresses of individual segments, e.g., data regions, code regions, and loader data structures of individual shared libraries, the heap, dynamically allocated memory areas, and the stacks of individual threads. Knowing neither the absolute addresses nor the offsets between individual regions, adversaries must rely on either just-in-time exploit construction [Snow et al. 2013] or some form of information leak or side channel to infer the addresses.

For code, randomization of the base address of the code region is often not enough, as adversaries can recover the locations of all functions with a single leaked code pointer if they know the relative offset to the desired function. For a single distribution or operating system, all installations generally share the same files, which allows adversaries to learn such offsets. Fine-grained code randomization schemes [Kil et al. 2006, Hiser et al. 2012, Wartell et al. 2012, Bigelow et al. 2015, Crane et al. 2015, Crane et al. 2015, Larsen et al. 2014] address the problem of well-known intra-file offsets by diversifying files on a per-system or per-process basis by permuting functions and basic blocks to increase the randomness inside single code regions. These systems introduce fine-grained diversity between individual instances of the program and may dynamically rediversify at specific intervals. Data Space Randomization (DSR) [Bhatkar and Sekar 2008, Cowan et al. 2003] increases diversity for data, changing the data or pointer representation dynamically by either encrypting the pointer or introducing an additional layer of indirection to hide the actual data regions from the adversary.

Stack canaries [Hiroaki and Kunikazu 2001] are an early attempt at stack integrity that relies on confidentiality of the canary. Random values are placed before return addresses on the stack. Stack canaries are integrity checked before the function returns to the caller. If an adversary uses a stack-based continuous buffer overflow [One 1996] to hijack control flow, then the canary is corrupted and the check before the return will fail. Unfortunately, it does not protect against direct targeted overwrites of the return address.

All randomization-based defenses are prone to information leaks; if an adversary can leak and infer the location or value of the protected data, then the defense can be bypassed. Randomization-based defenses are usually an additional step the adversary needs to compromise, making the attack harder and less likely to succeed. Recently, several attacks have shown that information hiding only gives limited security and adversaries can often recover the target addresses if a larger piece of data (such as a stack or metadata table) is hidden in the virtual address space [Göktaş et al. 2016, Oikonomopoulos et al. 2016, Evans et al. 2015a].

1.4 Data-Flow and Control-Flow Integrity

Data-flow and control-flow integrity both ensure that data values are valid. Compared to integrity, flow integrity ensures the security property not when data is written but when data is read. Every time a data value is used, a flow integrity check ensures that the value is benign.

Instead of preventing data corruption, Data-Flow Integrity (DFI) [Castro et al. 2006] detects corruption by checking read instructions. This prohibits corrupted

data from being used in computations but allows data in the process to be corrupted through memory safety violations. An adversary can use a corrupted pointer to modify a data value, but the corrupted value cannot be used in a later computation. DFI enforces reaching definitions for data values. Only program locations that are supposed to write a data value pass the check whenever data is read. DFI records the location for each memory write. For each read operation, DFI encodes the set of benign write locations and, at runtime, checks if the last write to that location came from such a benign location. In program terms, when reading the `isAdmin` flag, the read check ensures that the value was last written from a code location that is allowed to write to that address. By delaying the check until the read instruction, DFI can achieve lower overhead than memory safety. Compared to WIT, DFI protects read instructions instead of write instructions. DFI shares the same limitations due to a similar points-to analysis and over-approximation of merged labels.

Control-Flow Integrity (CFI) detects control-flow hijacking attacks by limiting the targets of control-flow transfers. Since the initial idea for the CFI defense mechanism [Abadi et al. 2005a] and the first (closed source) prototype were presented in 2005, a plethora of alternate CFI-style defenses have been proposed and implemented (see Burow et al. [2016], Burow et al. [2017] for a survey). Any CFI mechanism consists of two abstract components: the often static analysis component that recovers the Control-Flow Graph (CFG) of the application with different levels of precision and the dynamic enforcement mechanism that restricts control flows according to the generated CFG (see Figure 1.2).

Listing 1.2 shows a simple program with five functions. The `foo` function uses a function pointer, and the CFI mechanism injects both a forward-edge (e.g., indirect function call) and a backward-edge (function return) check. The function pointer points to either `bar` or `baz`. Depending on the forward-edge analysis, different sets of targets are allowed at runtime.

Figure 1.2 A CFI mechanism restricts control-flow hijacking to targets that are valid according to a pre-determined control-flow graph (black arrows). All other targets are rejected (red arrows).

```
void aa();
void ab();
void ac();
void ad(int, int);

void foo(int usr) {
  void (*func)();

  // func points to either bar or baz
  if (usr == MAGIC)
    func = aa;
  else
    func = ac;

  // forward-edge CFI check
  // depending on the precision of CFI:
  // a) all functions {aa, ab, ac, ad, foo} are valid
  // b) functions with prototype "void (*)()" are valid
  //    i.e., {aa, ab, ac}
  // c) only address-taken functions are valid, i.e., {aa, ac}
  CHECK_CFI_FORWARD(func);
  func();

  // backward-edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Listing 1.2 Simple program that showcases different granularities of precision for CFI mechanisms. Depending on the precision of the static analysis, only a subset of targets is valid.

For forward edges, the CFG generation enumerates all possible targets, often based on information from the source language such as symbol information, function prototypes, or class inheritance information. Switch statements in C/C++ are a good example as the different targets are statically known and the compiler can generate a fixed jump table and emit an indirect jump with a bound check to guarantee that the target used at runtime is one of the valid targets in the switch statement. For indirect function calls through a function pointer, the analysis becomes more complicated as the target may not be known a priori. Common source-based analyses use a type-based approach and, looking at the function prototype of the function pointer that is used, enumerate all matching functions. Different CFI mechanisms use different forms of type equality and use, e.g., any defined function, functions with the same arity (number of arguments), or functions with the same signature

(arity and equivalence of argument types) to distinguish valid call targets. At runtime, any function with matching signature is allowed.

Just looking at function prototypes likely yields several collisions where functions are defined that may never be called in practice. The analysis therefore over-approximates the valid set of targets. In practice, the compiler can check which functions are *address taken*, i.e., there is a source line that generates the address of the function and stores it. The CFI mechanism may reduce the number of allowed targets by intersecting the sets of equal function prototypes and the set of address-taken functions. For virtual calls, i.e., indirect calls in C++ that depend on the type of the object and the class relationship, the analysis can further leverage the type of the object to restrict the valid functions, e.g., the default constructors of all classes have the same signature but only the constructors of the subset of related classes are feasible.

The constructed CFG for the forward edge is stateless, i.e., the *context of the execution* is not considered and each control-flow transfer is independent of all others. On one hand, at runtime only one target is allowed for any possible transfer, namely, the target address currently stored at the memory location of the code pointer. CFG construction, on the other hand, over-approximates the number of valid targets with different granularities, depending on the precision of the analysis. Some mechanisms take path constraints into consideration [Niu and Tan 2015, van der Veen et al. 2015] and check (for a limited depth) if the path taken through the application is feasible by using a dynamic analysis approach that validates the current execution [van der Veen et al. 2015]. So far, only a few mechanisms look at the path context as this incurs dynamic tracking costs at runtime.

CFI can be enforced at different levels. Sometimes the analysis phase (CFG construction) and enforcement phase even overlap [van der Veen et al. 2015, Payer et al. 2015c], for instance, when considering path constraints. Most techniques have two fundamental mechanisms, one for forward-edge transfers and one for backward-edge transfers. Figure 1.2 shows how CFI restricts the set of possible target locations by executing a runtime monitor that validates the target according to the constructed set of allowed targets. If the observed target is not in that set, the program terminates. For forward-edge transfers, the code is often instrumented with some form of equivalence check. The check ensures that the target observed at runtime is in the set of valid targets. This can be done through a full set check or a simple type comparison that, e.g., hashes function prototypes and checks if the hash for the current target equals the expected hash at the call site. The hash for the function can be embedded inline in the function, before the function, or in an orthogonal metadata table.

Most mechanisms for the forward edge are stateless and allow an attacker to redirect control flow to any valid location as identified by the CFG analysis. Limiting the size of the target sets constrains the attacker on the forward edge. Future extensions of CFI should become context sensitive to increase the protection guarantees for the forward edge. If implemented correctly (i.e., considering language-specific information to increase precision [Schuster et al. 2015] and building the analysis on high-level data available at compile time), CFI is an efficient mitigation that constrains the control flow of the process, protecting against control-flow hijacking. Adversaries may still corrupt memory and data-only attacks are still in scope. For the forward edge, a strong mechanism must consider language-specific semantics to restrict the set of valid targets as much as possible.

Backward-edge transfers are harder to protect than forward-edge transfers as they require context sensitivity to be effective. When using a stateless, context-insensitive approach, the attacker may redirect the control flow to any valid call site when returning from a function [Carlini and Wagner 2014, Davi et al. 2014, Göktas et al. 2014a, Carlini et al. 2015e]. This imprecision often leaves enough targets for an attacker. The backward edge should therefore be protected through a stack integrity mechanism like a shadow stack or safe stack.

1.5 Policy Enforcement

Security defenses rely on a set of assumptions about the generated code and the runtime system. Most assumptions are only encoded indirectly (if at all) and not enforced throughout the software stack. Due to the complexity of the overall software stack, formal verification of components is restricted to high assurance domains with notable exceptions, including the CompCert verified compiler and the seL4 verified kernel, efforts that cannot be repeated for all software.

Many defenses rely on immutable code and non-executable memory regions. These two constraints can be enforced by hardware at the page table level but do not always hold due to exceptions. Another constraint that many code-reuse defenses depend on are immutability of vtables, jump tables, and other sensitive data. Even if the defense ensured that the assumption held at the compiler level where the initial transformation/instrumentation was carried out, there are no guarantees that the assumption remains enforced at runtime. Any software along the stack, compiler optimizations, assembler, linker, runtime loader, or operating system, may break the assumption. For example, the dynamic loader may copy read-only data into a writable section at runtime that then remains writable [Ge et al. 2017], favoring compatibility and performance over security. If a vtable of a class is defined

in a library but the executable implements the constructor, the loader relocates the vtable to a writable data section in the executable at runtime. An attacker may then freely modify the actual vtable code pointers while defense mechanisms like CFI will verify the correctness of the underlying vtable pointer. Another example is a compiler optimization that removes an inserted security check based on the assumption of certain language semantics or a certain memory model. While the compiler is free to do so according to the programming language, an attacker need not adhere to the semantics of the programming language [D'Silva et al. 2015].

Due to compatibility and performance trade-offs, full memory safety is rarely used in practical systems. As other defenses are not complete, allowing either remaining attack vectors or partial attacks, secondary defenses are needed. In addition, layering defenses can decrease the trust needed in a defense mechanism. Secondary defenses like intrusion detection, logging, auditing, or system call monitors allow a second line of defense to detect if a program is compromised.

System call monitors [Provos 2003, Goldberg et al. 1996, Alexandrov et al. 1999, Cowan et al. 2000, Acharya and Raje 2000, Bauer 2006, Fetzer and Suesskraut 2008, Watson et al. 2010] enforce a per-application policy that limits the damage a compromised process can do to the system, e.g., limiting the commands that can be executed and files that can be accessed. Such monitors can be implemented through various enforcement mechanisms, from in-kernel solutions [Wright et al. 2002] to ptrace-based solutions.

1.6 An Adversary's Toolkit

Adversaries have a clear target in mind when designing attacks. The attack usually follows the simplest path that achieves the desired target under the constraints of the adversary. The goals of the adversary are either to leak information or to gain certain privileges on a system.

Ideally, memory safety (Section 1.1) stops all attacks by detecting or protecting against the initial memory safety violation. As a first step, the adversary needs to modify the process state, leveraging the memory safety violation to corrupt code, code pointers, data, or data pointers. At the level of memory safety, the adversary does not have any capabilities to compromise the process state because the attack is detected when an invalid pointer is dereferenced for reading or writing. Defenses that target the integrity of code or data (Section 1.2) stop the attack at this level. Having privileges to read or write arbitrary data is not enough; due to the large virtual address space, defenses can "hide" sensitive or confidential data by shuffling and randomizing their locations and values (Section 1.3). Defenses that protect

data-flow integrity can still stop an attack despite a powerful adversary that has the ability to change data and knows the addresses of the targets. Data-flow integrity stops the attack whenever the modified data is used. For example, a defense may stop the process whenever a modified code pointer is dereferenced and used to transfer control (Section 1.4). If the attack is not stopped by a defense, then the adversary may execute arbitrary computation in the compromised process. Security policies at all these levels may be enforced through different techniques, relying on either modified hardware, a compiler, or binary rewriting (Section 1.5). To make permanent changes and to interact with the environment, the adversary must rely on system calls or memory-mapped files. Auxiliary or high-level defenses evaluate the state of a process externally to detect anomalies. For example, a defense may check system calls and arguments used by the process to decide, for each system call, if it is in the set of allowed system calls that this program may execute.

1.6.1 Code Corruption

Code corruption allows an adversary to control what code is executed. An adversary may inject new code or modify existing code to change the computation of the program to an arbitrary adversary-controlled computation. Code corruption and code injection were the most common attack vector due to its relative simplicity until the enforcement of code integrity. If the adversary has access to an executable code region and control over the instruction pointer, code corruption is the most straightforward attack vector. Code corruption allows the adversary to gain full control over all memory in the process and to execute arbitrary system calls on behalf of the process [One 1996, Durden 2002, Butler and Anonymous 2004]. See Figure 1.3 for an example of a stack smashing attack that redirects control flow to injected code in a stack-based buffer. Note that this attack assumes either that no stack-based defenses or code injection defenses are present or that they can be circumvented.

Modern systems enforce a separation between code and data regions through an executable bit on a per-page basis. Code corruption modifies memory that is mapped as executable, thereby changing existing instructions or inserting new instructions into the executing program. Code injection adds new code to the process memory by either remapping a data region as executable, writing data to an executable region, or forcing a just-in-time code generator to emit specific code in an executable area. For a successful code corruption attack, the adversary must both circumvent code integrity and recover certain addresses, e.g., the location where code should be emitted.

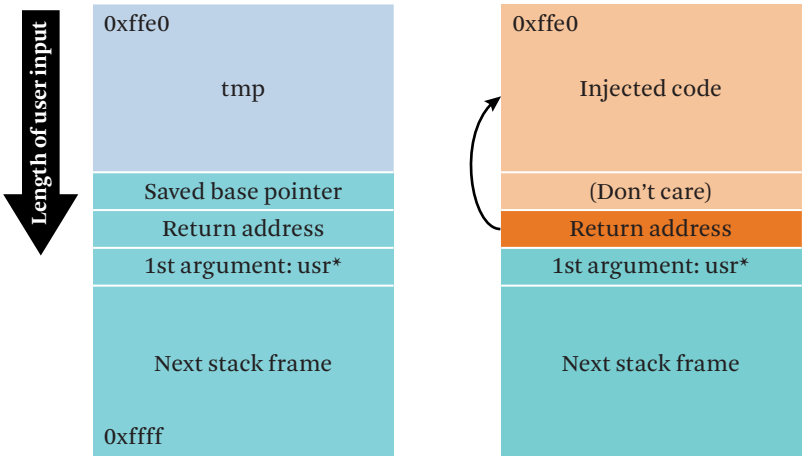

Figure 1.3 Straightforward stack-based code injection and control-flow hijacking through the stored return instruction pointer.

1.6.2 Control-Flow Hijacking

Control-flow hijacking happens whenever an adversary manages to redirect control flow of the program from expected locations (i.e., following control flow as defined through the program's control-flow graph) to an adversary-controlled location. Common architectures like ARM, x86, x86-64, or MIPS allow two different ways to transfer control (branch) from one location in memory to another location. On the instruction level, control-flow transfers are encoded either direct or indirect. For direct branches, the target is usually a relative distance from the current instruction pointer, and the distance is encoded as an immediate offset in the instruction itself (e.g., `0x75 0x02` encodes a `jne` instruction–jump if the zero bit in the flags is 0—which transfers control two bytes forward in execution flow for x86). Under the assumption of code integrity, i.e., code remains immutable throughout the execution of the program, an adversary cannot modify the target of these instructions as the immediate offset is encoded as part of the immutable instruction itself. While the target of the branch is immutable, an adversary may control the data that is used in the comparison to set the flags and thereby influence whether or not the branch is taken for conditional branches.

Indirect branches allow transfer of control flow to an arbitrary location in memory. The target is usually encoded as an absolute address to a code location, and the indirect control-flow transfer instruction dereferences a memory location or uses the value in a register (e.g., `0xff 0xd0` encodes an indirect call through the eax/rax

register on x86/x86-64). An adversary with knowledge of the location of the code pointer may modify the target through an arbitrary memory write. Many platforms have several types of indirect control-flow transfers for different use cases. For example, function returns (`ret` on x86) transfer control indirectly to the address that is at the top of the stack, indirect jumps (`jmp` on x86) transfer control to the specified address in a register or memory location, and indirect calls (`call` on x86) push the current instruction pointer (after the call) to the stack and execute an indirect jump. Indirect control-flow transfers are further divided into *forward-edge* control-flow transfers and *backward-edge* control-flow transfers. Forward-edge control-flow transfers direct control flow forward to a new location and are used in indirect jump and indirect call instructions, which are mapped at the source code level to switch statements, indirect calls, or virtual calls. The backward edge is used to return to a location that was used in a forward-edge transfer earlier, i.e., when returning from a function call. For simplicity, we do not discuss interrupts, interrupt returns, and exceptions in detail.

Memory corruption allows the adversary to modify data outside of the program semantics defined by the programmer. In programs, code pointers, data pointers, sensitive data, and other data are not separated and they are all accessible through the same address space. By changing program semantics, an adversary can use any memory write to change a code pointer. The only isolation between different data types is enforced by the semantics defined through the programmer. If the program is buggy, these semantics may break down and allow arbitrary changes in semantics, leading to so-called weird machines [Bratus et al. 2011]. For example, the programmer may have intended to change a data value on the stack, but due to a miscalculation through an offset that is under the adversary's control, the memory write ends up corrupting a code pointer on the process's heap. In control-flow hijack attacks, adversaries compromise a pointer, make it point to a memory location that contains a code pointer, and corrupt the original code pointer with a target address. When the program follows the corrupted code pointer, the control flow is redirected from a benign intended program location to a location under the adversary's control and the control-flow hijack attack is complete.

1.6.3 Code Reuse

Adversaries use control-flow hijacking to redirect the control flow of the program to alternate executable locations. With the widespread use of defenses enforcing data execution prevention [Andersen and Abella 2004], adversaries had to adjust attacks to achieve code execution. Whenever code injection is infeasible,

the attacker must resort to code reuse. In a code-reuse attack, the adversary repurposes existing code in alternate ways. The existing code of the application is analyzed and broken into small components called *gadgets*. Gadgets usually end in an indirect control-flow transfer (e.g., a return, indirect call, or indirect jump instruction) to allow the adversary to redirect execution to the next gadget. Gadgets can be stitched together in arbitrary ways, allowing adversaries to inject Turing-complete programs by modifying code pointers and supplemental data.

Over time, different flavors of code reuse have been developed. Starting with return to libc [Wojtczuk 1998], privileged functions like mmap or mprotect were reused to remap injected data as executable. Return-Oriented Programming (ROP) [Roemer et al. 2012] generalized return to libc attacks to Turing-complete ROP payloads, and Jump-Oriented Programming (JOP) [Checkoway et al. 2010, Bletsch et al. 2011] finally moved from return instructions to indirect jump or call instructions to transfer control between individual gadgets. In response to advances in defense techniques, attacks have been refined over the last couple of years [Schuster et al. 2015, Evans et al. 2015, Carlini et al. 2015e, Snow et al. 2013], becoming increasingly powerful. Figure 1.4 shows a stack-based ROP attack. Note that the simplified example assumes that no stack-based defenses are in place but code integrity protects against code injection in a stack-based buffer. In addition, the attacker needs to know or learn the locations of the desired gadgets.

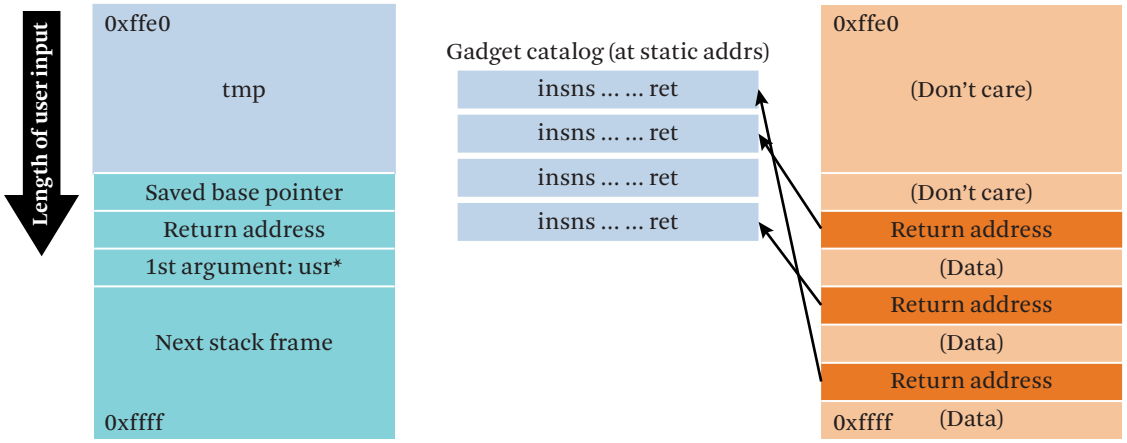

Figure 1.4 Stack-based ROP attack, redirecting control flow to a set of gadgets. This example assumes x86 calling conventions where all parameters are passed on the stack.

1.6.4 Data-Only Attacks

Data-only attacks exclusively modify data that is never loaded into the program counter of the CPU, excluding all code pointers. Data-only attacks may write to data that is used in comparisons or control-flow decisions, but the modified data never encodes a target address for the program counter. Data-only attacks can be realistic threats [Chen et al. 2005] and allow adversaries to compromise systems without hijacking the control flow, simply bending it along the valid control-flow graph of the program [Carlini et al. 2015e, Evans et al. 2015, Hu et al. 2015]. By modifying data, an adversary may still influence the control flow of the application. For example, redirecting control flow to an if-branch instead of an else-branch by modifying the tested condition through a memory corruption causes a trace through the code that is not intended by the programmer.

Data-only attacks can be grouped into two different classes of increasing complexity: non-control-data attacks and control-flow bending attacks. *Non-control-data attacks* modify a sensitive data area like a flag or string. For example, an attack could compromise the string that is passed to an `exec()` system call, enabling privileged mode for a JavaScript interpreter that allows unprivileged code from the web to execute with local privileges, or change the user id of a process in the kernel to 0 (root). These are realistic threats that exploit a program that enforces different privileges for individual components or restricts privileges inside the program itself. *Control-flow bending attacks* increase the sophistication of data-only attacks by "bending" the control flow along valid edges in the control-flow graph across multiple basic blocks. In the short C example below, assuming that no benign execution would allow `usr` to point to the third argument `a`, a control-flow bending attack could force, through the memory error in line 6, both of the if-branches to execute.

```
1  void vulnerable(int *usr, int usr2, int a)
2    if (a) {
3      .
       .
       .
4    }
5    // memory corruption
6    *usr = usr2;
7    if (!a) {
8      .
       .
       .
9    }
10 }
```

In control-flow bending attacks, the adversary uses the side effects of the executed basic blocks to modify the program state. The memory safety violation allows an initial setup of the program state that in turn allows steering the control flow along the path that executes the desired computations as side effects. For example, targets for modification could be interwoven data structures and pointers.

1.7 Conclusion

Memory safety violations continue to pose an important threat despite a long history of research. Existing solutions offer either complete protection but result in prohibitive performance overhead and compatibility issues, or partial protection with low overhead and high compatibility. Attacks can be stopped at several levels of abstraction, offering different trade-offs between defense compatibility, strength, and performance. Table 1.1 shows a summary of different defense mechanisms and their overhead, strength, and compatibility.

Current systems leverage only code integrity, ASLR, and stack canaries. The status quo protects against code injection attacks but lacks in protection against code-reuse or data-only attacks. These advanced attack vectors are still possible by circumventing probabilistic defenses like ASLR. In the near future, CFI and stack integrity will likely be deployed, building on code integrity and increasing the protection against code-reuse attacks. Going forward, we encourage the research community to develop more principled defenses against memory corruption that stop code-reuse attacks as well as data-only attacks.

Table 1.1 Summary of Defense Mechanisms and Corresponding Policies

Policy	Technique	Strength [a]	Performance Overhead [b]	Compatibility [b]	Weakness [c]
Spatial Memory Safety	SoftBound, CCured, Cyclone	Targeted	80%	High	Overhead
Temporal Memory Safety	CETS	Targeted	> 100%	High	Overhead
Type Safety	TypeSan, CaVer	Targeted	5–50%	High	Overhead
Data Integrity	Write Integrity Testing	High	10–25%	Composition	Over-approximation, unprotected reads, field protection
Code Integrity	Page flags	Partial	< 1%	High	Dynamically generated code
Return Integrity	Stack canaries	Low	< 5%	High	Information leaks, direct writes
Data Space Randomization	Data Space Randomization (DSR)	Partial	15–25%	Composition	Over-approximation, information leaks
Address Space Randomization	Address Space Layout Randomization	High	< 10%	High	Information leaks
Data-Flow Integrity	Data-Flow Integrity	Partial	100–200%	Composition	Over-approximation
Control-Flow Integrity	Control-Flow Integrity	Targeted	< 10%	Composition	Over-approximation

a. Strength lists the power of a mechanism: Targeted (completely stopping a type of memory corruption), High (mostly covering an issue), Partial (partially covering an issue), Low (somewhat hindering the adversary).

b. Performance overhead and compatibility list reported performance impact and potential compatibility issues: High (works for most software), Composition (does not work with shared libraries).

c. Weakness lists potential drawbacks of a policy.