Software Security

Principles, Policies, and Protection

Mathias Payer

July 2021, v0.37

Contents

1 Introduction

Browsing through any daily news feed, it is likely that the reader comes across several software security-related stories. Software security is a broad field and stories may range from malware infestations that abuse a gullible user to install malicious software to widespread worm outbreaks that leverage software vulnerabilities to automatically spread from one system to another. Software security (or the lack thereof) is at the basis of all these attacks.

But why is software security so difficult? The answer is complicated. Both *protection against* and *exploitation of* security vulnerabilities cross-cut through all layers of abstraction and involve human factors, usability, performance, system abstractions, and economic concerns. While adversaries may target the weakest link, defenders have to address all possible ***attack vectors***. An attack vector is simply an entry for an attacker to carry out an attack, i.e., an opening in the defenses allowing the attacker to gain unintended access. A single flaw is enough for an attacker to compromise a system while the defender must prohibit any feasible attack according to a given threat model. As an example, an attacker requires a single exploitable bug to compromise a system while the defender must prohibit the exploitation of all flaws that are present in a system. If you

compare this to logic formulas, it is much simpler to satisfy an "it exists" condition than a "for all condition".

Security and especially system and software security concerns permeate all areas of our life. We interact with complex interconnected software systems on a regular basis. Bugs or defects in these systems allow attackers unauthorized access to our data or enable them to escalate their privileges such as by installing malware. Security impacts everyone's life and it is crucial for a user to make safe decisions. To manage an information system, people across several layers of abstraction have to work together: managers, administrators, developers, and security researchers. A manager will decide on how much money to invest into a security solution or what security product to buy. Administrators must carefully reason about who gets which privileges. Developers must design and build secure systems to protect the integrity, confidentiality, and availability of data given a specific access policy. Security researchers identify flaws and propose mitigations against weaknesses, vulnerabilities, or systematic attack vectors.

Security is the application and enforcement of policies through defense mechanisms over data and resources. Security policies specify *what* we want to enforce. Defense mechanisms specify *how* we enforce the policy (i.e., an implementation/instance of a policy). For example, "Data Execution Prevention" is a mechanism that enforces a Code Integrity policy by guaranteeing each page of physical memory in the address space of a process is either writable or executable but never both. *Software Security* is the area of security that focuses on (i) testing, (ii) evaluating, (iii) improving, (iv) enforcing, and (v)

proving *security properties* of software.

To form a common basis of understanding and to set the scene for software security, this book first introduces and defines *basic security principles*. These principles cover confidentiality, integrity, and availability to define what needs to be protected, compartmentalization to understand approaches of defenses, threat models to reason about abstract attackers and behavior, and differences between bugs and vulnerabilities.

During the discussion of the *secure software life cycle*, we will evaluate the design phase with a clear definition of requirement specification and functional design of software before going into best implementation practices, continuous integration, and software testing along with secure software updates. The focus is explicitly not software engineering but security aspects of these processes.

Memory and type safety are the core security policies that enable semantic reasoning about software. Only ff memory and type safety are guaranteed then we can reason about the correctness of software according to its implementation. Any violation of these policies results in exceptional behavior that allows an attacker to compromise the internal state of the application and execute a so-called *weird machine*. Weird machines no longer follow the expected state transitions of an application as defined in source code as the application state is compromised. An execution trace of a weird machine always entails some violation of a core security policy to break out of the constraints of the application control-flow (or data-flow). The CPU only executes the underlying instructions. When the data is compromised, the high-level properties that the compiler

or runtime system assumed will no longer hold and therefore any checks that were removed due to these assumptions may no longer hold.

The section on *attack vectors* discusses different types of attacks. Starting with a confused deputy that abuses a given API to trigger a compartment into leaking information or escalating privileges to control-flow hijacking attacks that leverage memory and type safety issues to redirect an application's control-flow to attacker-chosen locations. Code injection and code reuse rewire the program to introduce attacker-controlled code into the address space of a program.

The *defense strategies* section lists different approaches to protect applications against software security violations. As new source code is written faster than it can be tested, some exploitable bugs will always remain despite the best development and testing efforts. As a last line of defense, mitigations help a system to ensure its integrity by sacrificing availability. Mitigations stop an unknown or unpatched flaw by detecting a policy violation through additional instrumentation in the program. During development, defense strategies focus on different approaches to verify software for functional correctness and to test for specific software flaws with different testing strategies. Sanitizers can help expose software flaws during testing by terminating applications whenever a violation is detected.

A set of case studies rounds off the book by discussing browser security, web security, and mobile security from a software and system security perspective.

This book is intended for readers interested in understanding

the status quo of software security, for developers that want to design secure software, write safe code, and continuously guarantee the security of an underlying system. While we discuss several research topics and give references to deeper research questions, this book is not intended to be research-focused but a source of information to dive into the software security area.

Disclaimer: this book is definitively neither perfect nor flawless. If you find spelling errors, language issues, textual mistakes, factual inconsistencies, or other flaws, please follow a responsible disclosure strategy and let us know by dropping us an email. We will happily update the text and make the book better for everyone.

Enjoy the read and ***hack the planet***!

2 Software and System Security Principles

The goal of software security is to allow any intended use of software but prevent any unintended use. Any unintended use may cause harm as in unintended use of compute resources outside of a defined allowed use. In this chapter, we discuss several system principles that are essential when building secure software systems. *Confidentiality*, *Integrity*, and *Availability* enable reasoning about different properties of a secure system. *Isolation* enforces the separation between components such that interaction is only possible along a well-defined interface that allows reasoning about access primitives. *Least privilege* ensures that each software component executes with the minimum amount of privileges. During *compartmentalization*, a complex piece of software is broken into smaller components. Isolation and compartmentalization play together, a large complex system is compartmentalized into small pieces that are then isolated from each other. The *threat model* specifies the environment of the software system, outlining the capabilities of an attacker. Distinguishing between *software bugs* and *vulnerabilities* helps us to decide about the risks of a given flaw.

2.1 Authentication

Authentication is the process of verifying if someone is who they claim to be. Through authentication, a system learns who you are based on what you know, have, or are. During authentication, a user verifies that they have the correct credentials. For example, to login on a system a user has to authenticate with their username and password. The username may either have to be entered or selected from a drop down list and the password has to be typed.

Authenticating through username and password is the most common form of verifying someones credential but alternate forms are possible too. Common classes of authentication forms are passwords (what you know), biometric (what you are), or demonstration of property (what you have).

Passwords are the classic approach towards authentication. During authentication, the user has to type their password or PIN (personal identification number) to login. The password is generally kept secret. Most systems provide usernames and password as the default authentication method. Limitations are the lack of replay resistance: an attacker that steals the raw biometric data can replay that data to authenticate as the user. This risk can be mitigated by a reasonable password update policy where, after a break, the users may be urged to update their passwords.

Another risk is that passwords can be brute-forced. During such a brute force attack, the attacker tries every single possible password combination. Over the years password policies have become highly restrictive and on some systems users have to

create new passwords every few months, they are not allowed to repeat passwords, and passwords must contain a set of unique character types (e.g., upper case letters, lower case letters, numbers, and special characters) to ensure sufficient entropy. Current best practices are to allow users freedom in providing sufficiently long passwords. It is easier to achieve good entropy with longer passwords than having users forget their complex short passwords.

Biometric logins may target fingerprints, Iris scans, or behavioral patterns (e.g., how you swipe across your screen). Using biometric factors for authentication is convenient as users cannot (generally) neither lose nor forget them. Their key limitation is the lack of replay resistance. Different to passwords, biometrics cannot be changed, so a loss of data means that this authentication form loses its utility. For example, if someone's fingerprints are known by the attacker, they can no longer be used for authentication.

Property can be anything the user owns that can be presented to the authentication system such as smartcards, smartphones, or USB keys. These devices have some internal key generation mechanism that can be verified. An advantage is that they are easily replaceable. The key disadvantage is that they should not be used by itself as, e.g., the smartphone may be stolen.

Instead of just using a single username and password pair, many authentication systems nowadays rely on two or more factors. For example, a user may have to log in with username, password, and a code that is sent to their phone via text message.

2.2 Access Rights

Access rights encode what entities a user has access to. For example, a user may be allowed to execute certain programs but not others. They may have access to their own files but not to files owned by another user. The Unix philosophy introduced a similar access right matrix consisting of user, group, and other rights.

Each file has an associated user which may have read, write, or execute rights. In addition to the user who is the primary owner, there may be a group with corresponding read, write, or execute rights, and all others that are not part of the group with the same set of rights. A user may be member of an arbitrary number of groups. The system administrator organizes group membership and may create new users. Through privileged services, users may update their password and other sensitive data.

More information about access rights, access control (both mandatory and discretionary) along with role based access control can be found in many books on Usenix system design or generally system security.

2.3 Confidentiality, Integrity, and Availability

Information security can be summarized through the three key concepts: confidentiality, integrity, and availability. The three concepts are often called the CIA triad. These concepts are sometimes called security mechanisms, fundamental concepts,

properties, or security attributes. While the CIA triad is somewhat dated and incomplete, it is an accepted basis when evaluating the security of a system or program. The CIA triad serves as a good basis for refinement and covers the core principles. *Secrecy* as a generic property ensures that data is kept hidden (secret) from an unintended receiver.

Confidentiality of a service limits access of information to privileged entities. In other words, confidentiality guarantees that an attacker cannot recover protected data. The confidentiality property requires authentication and access rights according to a policy. Entities must be both named and identified and an access policy determines the access rights for entities. Privacy and confidentiality are not equivalent. Confidentiality is a component of privacy that prevents an entity from viewing privileged information. For example, a software flaw that allows unprivileged users access to privileged files is a violation of the confidentiality property. Alternatively, encryption, when implemented correctly, provides confidentiality.

Note that confidentiality ensures that someone else's data is being kept secret For example, the OS ensures confidentiality of a process' address space by hiding it from other processes.

Integrity of a service limits the modification of information to privileged entities. In other words, integrity guarantees that an attacker cannot modify protected data. Similar to confidentiality, the integrity property requires authentication and access rights according to a policy. For example, a software flaw that allows unauthenticated users to modify a privileged file is a violation of the integrity policy. For example, a checksum that is protected against adversarial changes can detect tampering

of data. Another aspect of integrity is replay protection. An adversary could record a benign interaction and replay the same interaction with the service. Integrity protection detects replayed transactions. In software security, the integrity property is often applied to data or code in a process.

For example, the OS ensures integrity of a process' address space by prohibiting other processes from writing to it.

Availability of a service guarantees that the service remains accessible. In other words, availability prohibits an attacker from hindering computation. The availability property guarantees that legitimate uses of the service remain possible. For example, allowing an attacker to shut down the file server is a violation of the availability policy.

For example, the OS ensures availability by scheduling each process a "fair" amount of time, alternating between processes that are ready to run.

The three concepts build on each other and heavily interact. For example, confidentiality and integrity can be guaranteed by sacrificing availability. A file server that is not running cannot be compromised or leak information to an attacker. For the CIA triad, all properties must be guaranteed to allow progress in the system.

Several newer approaches extend these three basic concepts by introducing orthogonal ideas. The two most common extensions are *accountability* and *non-repudiation*, referring that a service must be accountable and cannot redact a granted access right or service. For example, a service that has given access to a file to an authorized user cannot claim after the fact that access

was not granted. Non-repudiation is, at its core, a concept of law. Non-repudiation allows both a service to prove to an external party that it completed a request and the external party to prove that the service completed the request.

Orthogonally, *privacy* ensures confidentiality properties for the data of a person. *Anonymity* protects the identity of an entity participating in a protocol.

Each property covers one separate aspect of information security. Policies provide concrete instantiations of any of the policies while mechanisms further refine a policy into an actual implementation. In practice, we will be working with policies that provide certain guarantees, following the core properties defined here. Policies themselves define the high level goals and the concrete mechanisms then enforce a given policy.

2.4 Isolation

Isolation separates two components from each other and confines their interactions to a well-defined API. There are many different ways to enforce isolation between components, all of them require some form of abstraction and a security monitor. The security monitor runs at higher privileges than the isolated components and ensures that they adhere to the isolation. Any violation to the isolation is stopped by the security monitor and, e.g., results in the termination of the violating component. Examples of isolation mechanisms include the process abstraction, containers, or SFI [33,34].

The *process abstraction* is the most well known form of isolation: individual processes are separated by the operating system from each other. Each process has its own virtual memory address space and can interact with other processes only through the operating system which has the role of a security monitor in this case. An efficient implementation of the process abstraction requires support from the underlying hardware for virtual memory and privileged execution. Virtual memory is an abstraction of physical memory that allows each process to use the full virtual address space. Virtual memory relies on a hardware-backed mechanism that translates virtual addresses to physical addresses and an operating system component that manages physical memory allocation. The process runs purely in the virtual address space and cannot interact with physical memory. The code in the process executes in non-privileged mode, often called user mode. This prohibits process code from interacting with the memory manager or side-stepping the operating system to interact with other processes. The CPU acts as a security monitor that enforces this separation and guarantees that privileged instructions trap into supervisor mode. Together privileged execution and virtual memory enable isolation. Note that similarly, a hypervisor isolates itself from the operating system by executing at an even higher privileged mode and mapping guest physical memory to host physical memory, often backed through a hardware mechanism to provide reasonable performance.

Containers are a lightweight isolation mechanism that builds on the process abstraction and introduces namespaces for kernel data structures to allow isolation of groups of processes. Normally, all processes on a system can interact with each

other through the operating system. The container isolation mechanism separates groups of processes by virtualizing operating system mechanisms such as process identifiers (pids), networking, inter process communication, file system, and namespaces.

Software-based Fault Isolation (SFI) [33,34] is a software technique to isolate different components in the same address space. The security monitor relies on static verification of the executed code and ensures that two components do not interact with each other. Each memory read or write of a component is restricted to the memory area of the component. To enforce this property, each instruction that accesses memory is instrumented to constrain the pointer to the memory area. To prohibit the isolated code from modifying its own code, control-flow transfers are carefully vetted and all indirect control-flow transfers must target well-known locations. The standard way to enforce SFI is to mask pointers before they are dereferenced (e.g., anding them with a mask: `and %reg, 0x00ffffff`) and by aligning control-flow targets and enforcing alignment.

Generally, lower levels of abstractions trust the isolation guarantees of higher levels. For example, a process trusts the operating system that another process cannot suddenly read its memory. This trust may be broken through side channels which provide an indirect way to recover (partial) information through an unintended channel. Threat models and side channels will be discussed in detail later. For now, it is safe to assume that if a given abstraction provides isolation that this isolation holds. For example, the process trusts the operating system (and the underlying hardware which provides privilege

levels) that it is isolated from other processes.

2.5 Least Privilege

The principle of least privilege guarantees that a component has the least amount of privileges needed to function. Different components need privileges (or permissions) to function. For example, an editor needs read permission to open a particular file and write permissions to modify it. Least privilege requires isolation to restrict access of the component to other parts of the system. If a component follows least privilege then any privilege that is further removed from the component removes some functionality. Any functionality that is available can be executed with the given privileges. This property constrains an attacker to the privileges of the component. In other words, each component should *only* be given the privilege it requires to perform its duty and no more. Note that privileges have a temporal component as well.

For example, a web server needs access to its configuration file, the files that are served, and permission to open the corresponding TCP/IP port. The required privileges are therefore dependent on the configuration file which will specify, e.g., the port, network interface, and root directory for web files. If the web server is required to run on a privileged port (e.g., the default web ports 80 and 443) then the server must start with the necessary privileges to open a port below 1024. After opening the privileged port, the server can drop privileges and restrict itself to only accessing the root web directory and its subdirectories.

2.6 Compartmentalization

The idea behind compartmentalization is to break a complex system into small components that follow a well-defined communication protocol to request services from each other. Under this model, faults can be constrained to a given compartment. After compromising a single compartment, an attacker is restricted to the protocol to request services from other compartments. To compromise a remote target compartment, the attacker must compromise all compartments on the path from the initially compromised compartment to the target compartment.

Compartmentalization allows abstraction of a service into small components. Under compartmentalization, a system can check permissions and protocol conformity across compartment boundaries. Note that this property builds on least privilege and isolation. Both properties are most effective in combination: many small components that are running and interacting with least privilege.

A good example of compartmentalization is the Chromium web browser. Web browsers consist of multiple different components that interact with each other such as a network component, a cache, a rendering engine that parses documents, and a JavaScript compiler. Chromium first separates individual tabs into different processes to restrict interaction between them. Additionally, the rendering engine runs in a highly restricted sandbox to limit any bugs in the parsing process to an unprivileged process.

2.7 Threat Model

A threat model is used to explicitly list all threats that jeopardize the security of a system. Threat modeling is the process of enumerating and prioritizing all potential threats to a system. The explicit motion of identifying all weaknesses of a system allows individual threats to be ranked according to their impact and probability. During the threat modeling process, the system is evaluated from an attacker's view point. Each possible entry vector is evaluated, assessed, and ranked according to the threat modeling system. Threat modeling evaluates questions such as:

- What are the high value-assets in a system?
- Which components of a system are most vulnerable?
- What are the most relevant threats?

As systems are generally large and complex, the first step usually consists of identifying individual components. The interaction between components is best visualized by making any data flow between components explicit, i.e., drawing the flow of information and the type of information between components. This first step results in a detailed model of all components and their interactions with the environment.

Each component is then evaluated based on its exposure, capabilities, threats, and attack surface. The analyst iterates through all components and identifies, on a per-component basis, all possible inputs, defining valid actions and possible threats. For each identified threat, the necessary preconditions are mapped along with the associated risk and impact.

A threat model defines the environment of the system and the capabilities of an attacker. The threat model specifies the clear bounds of what an attacker can do to a system and is a precondition to reason about attacks or defenses. Each identified threat in the model can be handled through a defined mitigation or by accepting the risk if the cost of the mitigation outweighs the risk times impact.

Let us assume we construct the threat model for the Unix "login" service, namely a password-based authentication service. Our application serves three use-cases: (i) the system can authenticate a user based on a username and password through a trusted communication channel, (ii) regular users can change their own password, and (iii) super users can create new users and change any password. We identify the following components: data storage, authentication service, password changing service, and user administration service according to the use-cases above.

The service must be privileged as arbitrary users are allowed to use some aspects of the service depending on their privilege level. Our service therefore must distinguish between different types of users (administrators and regular users). To allow this distinction, the service must be isolated from unauthenticated access. User authentication services are therefore an integral part of the operating system and privileged, i.e., run with administrator capabilities.

The data storage component is the central database where all user accounts and passwords are stored. The database must be protected from unprivileged modification, therefore only the administrator is allowed to change arbitrary entries

while individual users are only allowed to change their own entry. The data storage component relies on the authentication component to identify who is allowed to make modifications. To protect against information leaks, passwords are encrypted using a salt and one-way hash function. Comparing the hashed input with the stored hash allows checking equivalence of a password without having to store the plaintext (or encrypted version) of the password.

The authentication service takes as input a username and password pair and queries the storage component for the corresponding entry. The input (login request) must come from the operating system that tries to authenticate a user. After carefully checking if the username and password match, the service returns the information to the operating system. To protect against brute-force attacks, the authentication service rate limits the number of allowed login attempts.

The password changing service allows authenticated users to change their password, interfacing with the data storage component. This component requires a successful prior authorization and must ensure that users can only change their own password but not passwords of other users. The administrator is also allowed to add, modify, or delete arbitrary user accounts.

Such an authentication system faces threats from several directions, providing an exhaustive list would go beyond the scope of this book. Instead, we provide an incomplete list of possible threats:

- Implementation flaw in the authentication service allowing either a user (authenticated or unauthenticated) to

authenticate as another user or privileged user without supplying the correct password.
- Implementation flaw in privileged user management which allows an unauthenticated or unprivileged user to modify arbitrary data entries in the data storage.
- Information leakage of the password from the data storage, allowing an offline password cracker to probe a large amount of passwords[1]
- A brute force attack against the login service can probe different passwords in the bounds of the rate limit.
- The underlying data storage can be compromised through another privileged program overwriting the file, data corruption, or external privileged modification.

2.8 Bug versus Vulnerability

A "bug" is a flaw in a computer program or system that results in an unexpected outcome. A program or system executes computation according to a specification. The term "bug" comes from a moth that deterred computation of a Harvard Mark II computer in 1947. Grace Hopper noted the system crash in the operation log as "first actual case of bug being found", see 2.1, [10]. The bug led to an unexpected termination

[1] Originally, the `/etc/passwd` file stored all user names, ids, and hashed passwords. This world readable file was used during authentication and to check user ids. Attackers brute forced the hashed passwords to escalate privileges. As a mitigation, Unix systems moved to a split system where the hashed password is stored in `/etc/shadow` (along with an id) and all other information remains in the publicly readable `/etc/passwd`.

of the current computation. Since then the term bug was used for any unexpected computation or failure that was outside of the specification of a system or program.

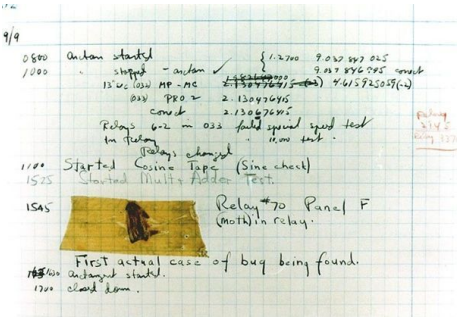


Figure 2.1: "First actual case of bug being found", note by Grace Hopper, 1947, public domain.

As a side note, while the term bug was coined by Grace Hopper, the notion that computer programs can go wrong goes back to Ada Lovelace's notes on Charles Babbage's analytical machine where she noted that "an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders."

A software bug is therefore a flaw in a computer program that causes it to misbehave in an unintended way while a hardware bug is a flaw in a computer system. Software bugs are due to human mistake in the source code, compiler, or runtime system. Bugs result in crashes and unintended program state. Software bugs are triggered through specific input (e.g., console input,

file input, network input, or environmental input).

If the bug can be controlled by an adversary to escalate privileges, e.g., gaining code execution, changing the system state, or leaking system information then it is called a vulnerability.

A vulnerability is a software weakness that allows an attacker to exploit a software bug. A vulnerability requires three key components (i) system is susceptible to flaw, (ii) adversary has access to the flaw (e.g., through information flow), and (iii) adversary has capability to exploit the flaw.

Vulnerabilities can be classified according to the flaw in the source code (e.g., buffer overflow, use-after-free, time-of-check-to-time-of-use flaw, format string bug, type confusion, or missing sanitization). Alternatively, bugs can be classified according to the computational primitives they enable (e.g., arbitrary read, arbitrary write, or code execution).

2.9 Summary

Software security ensures that software is used for its intended purpose and prevents unintended use that may cause harm. Security is evaluated based on three core principles: confidentiality, integrity, and availability. These principles are evaluated based on a threat model that formally defines all threats against the system and the attacker's capabilities. Isolation and least privilege allow fine-grained compartmentalization that breaks a large complex system into individual components where security policies can be enforced at the boundary between components

based on a limited interface. Security relies on abstractions to reduce complexity and to protect systems [17].

3 Secure Software Life Cycle

Secure software development is an ongoing process that starts with the initial design and implementation of the software. The secure software life cycle only finishes when software is retired and no longer used anywhere. Until this happens, software is continuously extended, updated, and adjusted to changing requirements from the environment. This setting results in the need for ongoing software testing and continuous software updates and patches whenever new vulnerabilities or bugs are discovered and fixed.

The environment such as operating system platforms (which can be considered software as well, following the same life cycle) co-evolve with the software running on the platform. An example is the evolution of security features available on the Ubuntu Linux distribution. Initially few to no mitigations were present but with each new release of the distribution, new hardening features are released, further increasing the resilience of the environment against unknown or unpatched bugs in the software. Ubuntu focuses on safe default configuration, secure subsystems, mandatory access control, filesystem encryption, trusted platform modules, userspace hardening, and kernel hardening. Together, these settings and changes make it harder for attackers to compromise a system.

Software engineering is different from secure software development. Software engineering is concerned with developing and maintaining software systems that behave reliably and efficiently, are affordable to develop and maintain, and satisfy all the requirements that customers have defined for them. It is important because of the impact of large, expensive software systems and the role of software in safety-critical applications. It integrates significant mathematics, computer science, and practices whose origins are in engineering.

Why do we need a secure software development life cycle? Secure software development focuses not only on the functional requirements but additionally defines security requirements (e.g., access policies, privileges, or security guidelines) and a testing/update regime on how to react if new flaws are discovered. Note, this is not a book on software engineering. We will not focus on waterfall, incremental, extreme, spiral, agile, or continuous integration/continuous delivery. The discussion here follows the traditional software engineering approach, leaving it up to you to generalize to your favorite approach. We discuss aspects of some modern software engineering concepts in a short section towards the end of this chapter.

3.1 Software Design

The design phase of a software project is split into two sub phases: coming up with a requirement specification and the concrete design following that specification. The requirement specification defines tangible functionality for the project, individual features, data formats, as well as interactions with the

environment. From a security perspective, the software engineering requirement specification is extended with a security specification, an asset identification, an environmental assessment, and use/abuse cases. The security specification involves a threat model and risk assessment. The asset specification defines what kind of data the software system operates on and who the entities with access to the data are (e.g., including privilege levels, administrator access, and backup procedures). Aspects of the environment are included in this assessment as they influence the threats, e.g., a public terminal is at higher physical risk than a terminal operating in a secured facility.

Transitioning from the requirement specification phase to the software design phase, security aspects must be included as integral parts of the design. This transition involves additional threat modeling based on the concrete design and architecture of the software system. The design of the software then extends the regular design documents with a concrete security design that ties into the concrete threat model. The actors and their abilities and permissions are clearly defined, both for benign users and attackers. During this phase, the design is reviewed from a functional but also from a security perspective to probe different aspects and to iteratively improve the security guarantees. The final design document contains full specifications of requirements, security constraints, and a formal design in prose.

3.2 Software Implementation

The implementation of the software project follows mostly regular software engineering best practices in robust programming [2]. Special care should be taken to ensure that source code is always checked into a source code repository using version control such as git or svn. Source version control systems such as GitHub allow the organization of source code in a git repository as well as corresponding documentation in a wiki. Individual flaws can be categorized in a bug tracker and then handled through branches and pull requests.

Each project should follow a strict coding standard that defines what "flavor" of a programming language is used, e.g., how code is indented and what features are available. For C++, it is worthwhile to define how exceptions will be used, what modern features are available, or how memory management should be handled. Alongside the feature definition, the coding standard should define how comments in the code are handled and how the design document is updated whenever aspects change. The Google C++ style guide or Java style guide are great examples of such specification documents. They define the naming structure in projects, the file structure, code formatting, naming and class interfaces, program practices, and documentation in an accessible document. Whenever a programmer starts on a project they can read the style guide and documentation to get a quick overview before starting on their component.

Similarly, newly added or modified source code should be reviewed in a formal code review process. When committing code

to a repository, before the new code is merged into the branch, it must be checked by another person on the project to test for code guidelines, security, and performance violations. The code review process must be integrated into the development process, working naturally alongside development. There are a myriad of tools that allow source review such as GitHub, Gerrit, and many others. For a new project it is important to evaluate the features of the different systems and to choose the one that best integrates into the development process.

3.3 Software Testing

Software testing is an integral component of software development. Each new release, each new commit must be thoroughly tested for functionality and security. Testing in software engineering focuses primarily on functionality and regression. Continuous integration testing, such as Jenkins or Travis, allow functional tests and performance tests based on individual components, unit tests, or for the overall program. These tests can run for each commit or at regular intervals to detect diversions quickly. While measuring functional completeness and detecting regression early is important, it somewhat neglects security aspects.

Security testing is different from functional testing. Functional testing measures if software meets certain performance or functional criteria. Security as an abstract property is not inherently testable. Crashing test cases indicate some bugs but there is no guarantee that a bug will cause a crash. Automatic security testing based on fuzz testing, symbolic execution, or

formal verification tests security aspects of the project, increasing the probability of a crash during testing. See Section 6.3 for more details on testing. Additionally, a red team evaluates the system from an adversary's perspective and tries to find exploitable flaws in the design or implementation.

3.4 Continuous Updates and Patches

Software needs a dedicated security response team to answer to any threats and discovered vulnerabilities. They are the primary contact for any flaw or vulnerability and will triage the available resources to prioritize how to respond to issues. Software evolves and, in response to changes in the environment, will continuously expand with new features, potentially resulting in security issues.

An update and patching strategy defines how to react to said flaws, how to develop patches, and how to distribute new versions of a software to the users. Developing a secure update infrastructure is challenging. The update component must be designed to frequently check for new updates while considering the load on the update servers. Updates must be verified and checked for correctness before they are installed. Existing software market places such as the Microsoft Store, Google Android Play, or the Apple Store provide integrated solutions to update software components and allow developers to upload new software into the store which then handles updates automatically. Google Chrome leverages a partial hot update system that quickly pushes binary updates to all Google Chrome instances to protect them against attacks. Linux distributions such as

Debian, RedHat, or Ubuntu also leverage a market-style system with an automatic software update mechanism that continuously polls the server for new updates and informs the user of new updates (e.g., through a pop up) or, if enabled, even automatically installs the security updates.

3.5 Modern Software Engineering

Software engineering processes underwent several improvements and many different management schemes exist. Under agile software development, one of those modern extensions, both requirements and solutions co-evolve as part of a collaborative team. The teams self-organize and restructure themselves depending on the changing requirements as part of the interaction with the customer. Under an agile system, an early release is constantly evaluated and further improved. The focus of agile development is on functionality and evolutionary planning.

This core focus on functionality and the lack of a written specification or documentation makes reasoning about security challenging. Individual team leads must be aware of security constraints and explicitly push those constraints despite them never being encoded. Explicitly assigning a member of the team a security role (i.e., a person that keeps track of security constraints) allows agile teams to keep track of security constraints and to quickly react to security relevant design changes. Every release under agile software development must be vetted for security and this incremental vetting must consider security implications as well (e.g., a feature may increase the threat surface or enable new attack vectors).

3.6 Summary

Software lives and evolves. The software development life cycle continues throughout the lifetime of software. Security must be a first class citizen during this whole process. Initially, programmers must evaluate security aspects of the requirement specification and develop a security-aware design with explicit notion of threats and actors. During the implementation phase programmers must follow strict coding guidelines and review any modified code. Whenever the code or the requirements change, the system must be tested for functionality, performance, and security using automated testing and targeted security probing. Last but not least, secure software development is an ongoing process and involves continuous software patching and updates – including the secure distribution of said updates.

4 Memory and Type Safety

A set of core security principles covers the security of system software. If these security principles hold then the software is secure. Memory safety ensures that pointers always point to a valid memory object, i.e., each memory access is in bounds and to a live object. Type safety ensures that objects are accessed with their corresponding types and casts observe the type hierarchy according to the true runtime type of an object. Under memory and type safety, all memory accesses adhere to the memory and type semantics defined by the source programming language. Bugs that cause the program to violate memory or type safety can be used to change the runtime state of the program. This modified runtime state leads to an execution that would not be possible under a benign execution of the program. For example, instead of encoding an image into a different format, an image program may connect to the internet and upload personal documents. Memory and type safety restrict the program behavior to what is specified in the source code. Bugs in the program logic may still allow arbitrary actions but no action that does not conform to a valid execution path in the program is possible.

For software written in high-level languages such as Java, core principles such as memory safety and type safety guarantee the

absence of low-level flaws that violate the high-level abstractions of the programming language and therefore limit possible attacks to bugs inherent to the program such as logic flaws. To be precise, memory and type safety limit attackers to a given specification and constraints of the implementation, not constraints of an underlying abstract machine. Memory unsafe languages like C/C++ do not enforce memory or type safety and data accesses can occur through stale/illegal pointers and an object may be reinterpreted under an illegal type.

The gap between the operational semantics of the programming language and the instructions provided through the underlying Instruction Set Architecture (ISA – e.g., the Intel x86 ISA defines the available instructions and their encoding on an x86 CPU) allow an attacker to step out of the restrictions imposed by the programming language and access memory out of context. If memory safety or type safety are violated, the program must no longer follow the well-defined control-flow graph and turns into a so-called weird machine [3]. You can think of a weird machine as a snapshot of the program state that was modified at a point in time. This modified memory snapshot may reuse the existing code sequences (e.g., individual instructions or short sequences of instructions) in unintended ways and out of context. Repurposing existing code snippets out of context turns a program into a weird machine. For example, in the code below, the weird machine could transfer control to `notcalled` by overwriting the function pointer `ptr` with the address of `notcalled` and the variable `flag` with a non-null value.

```
1  // this function is never called
2  void notcalled();
3
4  void addrtaken();
5
6  int flag = 0;
7  void (*ptr)() = &addrtaken;
8
9  void func() {
10   if (flag != 0) {
11     // under attack, this may call notcalled
12     ptr();
13   }
14 }
```

The need for memory or type safety checks depends on the programming language. Some languages inherently enforce memory and type safety (e.g., functional languages generally do not expose pointers to the programmer) and therefore do not require runtime checks. A low-level systems language such as C requires explicit checks to guarantee memory and type safety as the programmer is not required to add sufficient checks.

4.1 Pointer Capabilities

Pointers are unstructured addresses to memory and a way to reference data or code. A pointer has an associated type and a value, the address it points to. Under C/C++ pointer

arithmetic allows modification of a pointer through increments and decrements. The validity of the pointer is not enforced through the programming language but must be checked by the programmer. For example, after a pointer increment the programmer must ensure that the pointer still points into a valid array. When dereferencing, the programmer is responsible to guarantee that the pointed-to object is still valid.

Memory safety is a program property which guarantees that 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 [9,21]. 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 potentially the 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* of 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 is correct (according to the type system of the programming language) and 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 occurs when such an invalid pointer is dereferenced.

4.2 Memory Safety

Memory corruption, the absence of memory safety, is the root cause of many high-profile attacks and the foundation of a plethora of different attack vectors. Memory safety is a general property that can apply to a program, a runtime environment, or a programming language. A program is memory safe, if all possible *executions* of that program are memory safe. A runtime environment is memory safe, if all runnable programs are memory safe. A programming language is memory safe, if all expressible programs are memory safe. Memory safety prohibits, e.g., buffer overflows, NULL pointer dereferences,

use after free, use of uninitialized memory, or double frees. So while the C programming language is not memory safe, a C program can be memory safe if all possible executions of the C program enforce memory safety due to sufficient memory safety checks by the programmer.

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 by 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.

4.2.1 Spatial Memory Safety

Spatial memory safety is a property that ensures that all memory dereferences of an application are within the bounds of their pointer's valid objects. A pointer references a specific address in an application's address space. Memory objects are allocated explicitly by calling into the memory allocator (e.g., through `malloc`) or implicitly by calling a function for local variables. An object's bounds are defined when the object is allocated and a pointer to the object is returned. Any computed pointer to that object inherits the bounds of the object. Pointer arithmetic may change the pointer to outside the object. Only pointers that point inside the associated object may be

dereferenced. Dereferencing a pointer that points outside of the associated object results in a spatial memory safety error and undefined behavior.

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

```
1  char *c = (char*)malloc(24);
2  for (int i = 0; i < 26; ++i) {
3    // 1.) buffer overflow for i >= 24
4    c[i] = 'A' + i;
5  }
6  // 2.) violation through a direct write
7  c[26] = 'A';
8  c[-2] = 'Z';
9  // 3.) invalid pointers: OK if not dereferenced
10 char *d = c+26;
11 d -= 3;
12 *d = 'C';
```

This example shows a classic overflow where an array is sequentially accessed past its allocated length. The iterator moves past the end of the allocated object and as soon as the pointer is dereferenced (to write), memory safety is violated, corrupting an adjacent memory object. In the second case, memory safety is violated through a direct overwrite where the index points outside of the bounds of the object. The third case is fine as the invalid pointer is never dereferenced. The C standard allows pointers to become invalid as long as they are not used.

For example, a pointer can be incremented past the bounds of the object. If it is decremented, it may become valid again. Note that a pointer may only become valid again for spatial safety. If the underlying object has been freed, the pointer cannot become valid again.

4.2.2 Temporal Memory Safety

Temporal memory safety is a property that ensures that all memory dereferences are valid at the time of the dereference, i.e., the pointed-to object is the same as when the pointer was created. When an object is freed (e.g., by calling `free` for heap objects or by returning from a function for stack objects), the underlying memory is no longer associated to the object and the pointer is no longer valid. Dereferencing such an invalid pointer results in a temporal memory safety error and undefined behavior.

Various forms of temporal memory safety violations exist. After allocation, memory of an object can be read before it is written, returning data from the previously allocated object in that area. A stale pointer can be used after the underlying object has been returned to the memory allocator and even after that memory has been reused for a different object. Temporal memory safety violations happen if the underlying memory object was freed as shown in the following example:

```
1 char *c = malloc(26);
2 char *d = c;
3 free(d);
4 // violation as c no longer points to a valid
      object
5 c[23] = 'A';
```

4.2.3 A Definition of Memory Safety

Memory safety is violated if undefined memory is accessed, either out of bounds or the underlying memory was returned to the allocator. When evaluating memory safety, pointers become capabilities, they allow access to a well-defined region of allocated memory. A pointer becomes a tuple of address, lower bound, upper bound, and validity. Pointer arithmetic updates the tuple. Memory allocation updates validity. Dereference checks capability. These capabilities are implicitly added and enforced by the compiler. Capability-based memory safety enforces type safety for two types: pointer-types and scalars. Pointers (and their capabilities) are only created in a safe way. Pointers can only be dereferenced if they point to their assigned, still valid region.

4.2.4 Practical Memory Safety

In *Java*, memory safety is enforced by the programming language and the runtime system. The programming language replaces pointers with references and direct memory access is

not possible. There is no way to explicitly free and return data to the runtime system, memory is implicitly reclaimed through garbage collection. The runtime system enforces memory through additional checks (e.g., bounds checks) and leverages a garbage collector to passively reclaim unused memory. Note that Java also enforces type safety with explicit type safety checks.

For *Rust*, a strict type system and ownership implements memory and type safety. References are bound to variables and clear ownership protects against data races: single mutable reference or zero or more immutable references. Memory is reclaimed when variables go out of scope. Interestingly, many of these guarantees can be enforced by the compiler resulting in zero-cost abstractions.

Non-system functional languages such as Haskell or OCaml are oblivious to memory violations as they do not require the concept of references but pass data and control in other forms. See Section 6.2 for a short discussion on language-based security.

For *C/C++* there are two approaches to achieve memory safety: either removing unsafe features by creating a dialect or to protect the use of unsafe features through instrumentation.

Dialects extend C/C++ with safe pointers and enforce strict propagation rules. Cyclone [12] restricts the C programming language to a safe subset by limiting pointer arithmetic, adding `NULL` checks, using garbage collection for heap and region lifetimes for the stack (one of the inspirations for Rust's lifetimes), tagged unions to restrict conversions, splitting pointers into the

three classes normal, never `NULL`, fat pointers, and replacing `setjmp` (`setjmp` provides an archaic form of handling special cases, allowing the developer to record a return point and then jump to that point on demand) with exceptions and polymorphism. Cyclone enforces both spatial and temporal memory safety. CCured [23] follows a similar idea and introduces a pointer inference system to reduce the overhead of pointer tagging and pointer tracking. Similarly, modern C++ variants such as C++1X support a memory safe subset that uses references and strict ownership for memory objects to track lifetimes. Whenever only the safe subsets are used, C++ can be memory (and type) safe.

Protecting the use of unsafe features requires a runtime system to keep track of all live objects and pointers, associating bounds with each pointer and liveness with each memory object. For each pointer dereference a bounds and liveness check ensures that the memory access is valid. For pointer assignments, the pointer inherits the bounds of the assigned reference. SoftBound [21] is a compiler-based instrumentation to enforce spatial memory safety for C/C++. The general idea is to keep information about all pointers in *disjoint* metadata, indexed by pointer location. The downside of the approach is an overhead of 67% for SPEC CPU2006.

```
1  struct BankAccount {
2    char acctID[3]; int balance;
3  } b;
4  b.balance = 0;
5  char *id = &(b.acctID);
6  // Instrumentation: store bounds
7  lookup(&id)->bse = &(b.acctID);
8  lookup(&id)->bnd = &(b.acctID)+3;
9  // --
10 char *p = id;  // local, remains in register
11 // Instrumentation: propagate information
12 char *p_bse = lookup(&id)->bse;
13 char *p_bnd = lookup(&id)->bnd;
14 // --
15 do {
16   char ch = readchar();
17   // Instrumentation: check bounds
18   check(p, p_bse, p_bnd);
19   // --
20   *p = ch;
21   p++;
22 } while (ch);
```

The code example shows the instrumentation for SoftBound. Allocated memory is instrumented to return bounds (allocated on a per-pointer basis). Pointer assignment propagates bounds. Whenever the pointer is dereferenced for reading or writing, the bounds are checked.

CETS [22], an extension for SoftBound, enforces temporal

memory safety by storing validity for each object and pointer. CETS leverages memory object versioning. The code instrumentation allocates a unique version to each allocated memory area and stores this version in the pointer metadata as well. Each deallocation is instrumented to destroy the version in the object's memory area, causing the pointer and object version to become out of sync. Upon dereference, CETS checks if the pointer version is equal to the version of the memory object. There are two failure conditions: either the area was deallocated and the version is smaller (0) or the area was reallocated to a new object and the version is bigger. Both error conditions result in an exception and terminate the program.

The instrumentation and metadata can be carried out with different trade-offs regarding performance, memory overhead, and hardware extensions [5,15,20].

4.3 Type Safety

> Well-typed programs cannot "go wrong".
> (Robin Milner)

Type-safe code accesses only well-typed objects it is authorized to access. The literature groups type safety into different classes: strongly typed or weakly typed (with implicit type conversion). The type system can orthogonally be either static or dynamic. Despite a lot of research in type safety, C/C++ which are *not* type safe remain popular languages. Note that full type safety does not imply memory safety. The two properties are distinct. A C++ program can be type safe but not memory safe, e.g.,

an array index may point outside of the bounds of an array in a perfectly type safe program, resulting in a memory safety violation. Similarly, memory safety does not imply type safety as a `char *` array may be wrongly interpreted as an object of a specific type.

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 as having a different type. Casting is allowed along the inheritance chain. Upward casts (upcasts) move the type closer to the root object, the type becomes more generic, while downward casts (downcasts) specialize the object to a subtype. For C, the type lattice is fully connected, any pointer type can be cast to any other pointer types with the validity of the cast being the responsibility of the programmer.

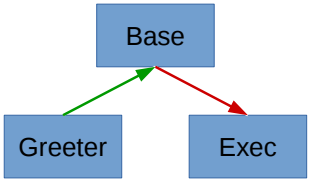
In C++ there are several casting operations. The most common ones are static and dynamic casts. A static cast `static_cast<ToClass>(Object)` results in a compile time check where the compiler guarantees that the type of `Object` is somehow related to `ToClass`, without executing any runtime check. A `dynamic_cast<ToClass>(Object)` results in a runtime check but requires Runtime Type Information (RTTI) and is only possible for polymorphic classes (i.e., they must have a vtable pointer in the object itself that uniquely identifies the class). Due to the runtime check, this type of cast results in performance overhead.

```
1  class Base { int base; };
2
3  class Exec: public Base {
4    public:
5      virtual void exec(const char *prg) {
6        system(prg);
7      }
8  };
9
10 class Greeter: public Base {
11   public:
12     int loc;
13     virtual void sayHi(const char *str) {
14       std::cout << str << std::endl;
15     }
16 };
17
18 int main() {
19   Base *b1 = new Greeter();
20   Base *b2 = new Exec();
21   Greeter *g;
22
23   g = static_cast<Greeter*>(b1);
24   g->sayHi("Greeter says hi!");
25
26   // Type confusion
27   g = static_cast<Greeter*>(b2);
28
29   // execute Exec::exec instead of Greeter::sayHi
30   // Low-level implementation: g[0][0](str);
31   g->sayHi("/usr/bin/xcalc");
32   g->loc = 12;  // memory safety violation
33
34   delete b1;
35   delete b2;
36 }
```

In the code example above, an object of type `Greeter` is allocated and then upcast to a `Base` type. Later, the Base type is downcast into `Exec`. As the runtime type of the object is `Greeter`, this downcast is illegal and results in type confusion – a violation of type safety.

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 complex set of type cast operations. Static casts are only checked at compile time to check if the two types are compatible. Dynamic casts execute a slow runtime check, which is only possible for polymorphic classes with virtual functions as 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, the figure below 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 explicitly 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++ due to the generalicity of the underlying type system and the opportunity to handle raw memory. Defenses therefore have to resort to runtime




```
Greeter *g = new Greeter();
Base *b = static_cast<Base*>(g);
Exec *e = static_cast<Exec*>(b);
```

Example of a type confusion vulnerability due to an illegal downcast.

checks. By making all casts in the program explicit and checking them for correctness at runtime, we ensure that the runtime type conforms to the statically assumed type at compile time [8,11,18,28]. Such a solution must keep metadata for all allocated memory objects, similarly to memory safety. Instead of bounds, a type safety mechanism records the true type of each allocated object. All cast types in C++ are then replaced with a runtime check.

4.4 Summary

Memory and type safety are the root cause of security vulnerabilities. Memory safety defines spatial and temporal capabilities for pointers. Spatial memory safety guarantees that pointers can only access objects in the corresponding bounds. Temporal memory safety checks for liveness of the underlying object.

When both spatial and temporal memory safety are enforced then a pointer is locked to a given memory object and can only dereference the area inside the object as long as that object is valid. Type-safe code accesses only the memory locations it is authorized to access. Type safety ensures that each object is only used with its correct type.

5 Attack Vectors

Understanding the intention of an attack helps with assessing the attack surface of a program. Not all attack vectors are feasible for all attacks and not all bugs allow instantiating all attack vectors. Attacker goals can be grouped into three broad classes: Denial of service (DoS); leaking information; and escalation of privileges, with confused deputies a special form of privilege escalation.

5.1 Denial of Service (DoS)

Denial of Service violates the *availability* property. DoS prohibits the legitimate use of a service by either causing abnormal service termination (e.g., through a segmentation fault) or overwhelming the service with a large number of duplicate/unnecessary requests so that legitimate requests can no longer be served. DoS also applies to outside the server setting, e.g., by corrupting a checksum of an image file it will no longer be displayed by an image viewer. This is the easiest attack to achieve as a server simply needs to be overwhelmed with bad requests to drown any legit requests.

5.2 Information Leakage

Information leakage violates the *confidentiality* property. *Information leaks* are abnormal or unintended transfers of sensitive information to the attacker. An information leak abuses an illegal, implicit, or unintended transfer of information to pass sensitive data to the attacker who should not have access to that data. An attacker can abuse an intended transfer of information and trick the program into sending unintended information (e.g., instead of sending a benign file as intended the server returns a privileged file).

Information leaks may be related to memory safety issues or logic errors. If the information leakage is due to a logic error, then the application can be tricked, following a benign path of execution, to leak the information to the attacker. Some information leaks are due to debug statements in the code, e.g., stack trace information that is returned to the user in the case of an exception or crash in a web application or publicly readable log files that record errors and crashes of applications together with auxiliary debug information such as crash addresses.

Memory or type safety violations may be used to leak information. For such leaks, the software flaw corrupts program state to replace or augment benign information with sensitive information. Leaking runtime information of an address space such as pointers, library, stack, or code locations enables bypassing probabilistic defenses such as Stack Canaries or Address Space Layout Randomization as these locations or values are only randomized on a per-process basis and are constant throughout the lifetime of a process.

5.3 Confused Deputy

A *confused deputy* is a kind of privilege escalation that tricks a component to execute an unprivileged action with higher privileges. A privileged component is supposed to only use its privileges for a benign action. By carefully setting up its input, an unprivileged attacker can make a privileged component (i.e., the confused deputy) *unintentionally* execute a privileged action (from the viewpoint of the developer). The confused deputy acts on behalf of the malicious component. The malicious component tricks the confused deputy into abusing its privileges on behalf of the attacker. The name "confused deputy" goes back to Barney Fife, an easily confused deputy who could be tricked into abusing his power in "The Andy Griffith Show" in the 1950s, see Figure 5.1.

The classic example of a confused deputy is a compiler that overwrites its billing file. On old mainframe computing systems customers had to pay for each run of the compiler due to the resources that were used on the mainframe for the compilation process. The compiler had access to a log file that recorded which user invoked the compiler. By specifying the log file as output file, the compiler could be tricked to overwrite the billing file with the executable program. The compiler required access to the log file (to record users) and the user invoking the compiler had the right to specify an output file to specify the new executable. As this compiler did not check if the output file was a special file, it acted as confused deputy. Another, more modern example is Cross-Site Scripting (XSS) which tricks a webpage to execute malicious JavaScript code in the user's browser.

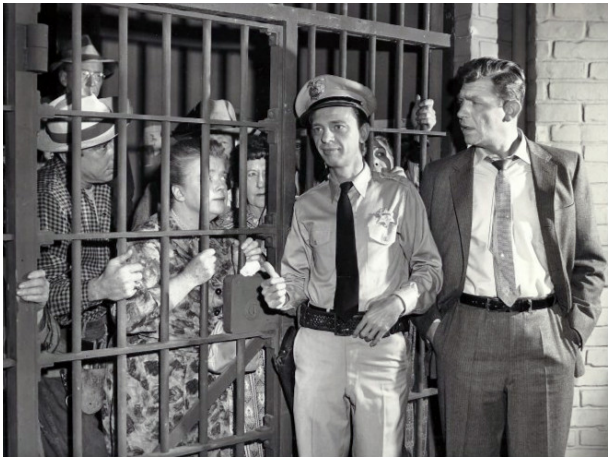


Figure 5.1: Barney Fife, the confused deputy, locks up half the town due to a chain of misunderstandings. From "The Andy Griffith Show", 1961, public domain.

5.4 Privilege Escalation

Privilege escalation is an *unintended* increase of privileges (from the viewpoint of the developer). An example of privilege escalation is gaining arbitrary code execution. Starting from access to the service and constrained to the functions provided by the service, the attacker escalates to arbitrary code execution where she has full access to all files, privileges, and system calls of that service. Another example of privilege escalation is a user without privileges that can modify files owned exclusively by the administrator, e.g., through a misconfiguration of the web interface.

While there are several forms of privilege escalation, we will focus on privilege escalation based on memory or type safety violations. Every such attack starts with a memory or type safety violation. Spatial memory safety is violated if an object is accessed out of bounds. Temporal memory safety is violated if an object is no longer valid. Type safety is violated if an object is cast and used as a different (incompatible) type. Any of these bug types allow a reconfiguration of program state that can trigger a privilege escalation when the legitimate code of the application acts on the corrupted state. In software security, these violations are used to *hijack control-flow*. The control flow is redirected to *injected code* or *existing code that is reused in unintended ways*. Alternatively, they can be used to *corrupt data*. Figure 5.2 shows the different paths an attack can take with the different mitigations that need to be circumvented along the attack flow for code corruption and control-flow hijacking attacks.

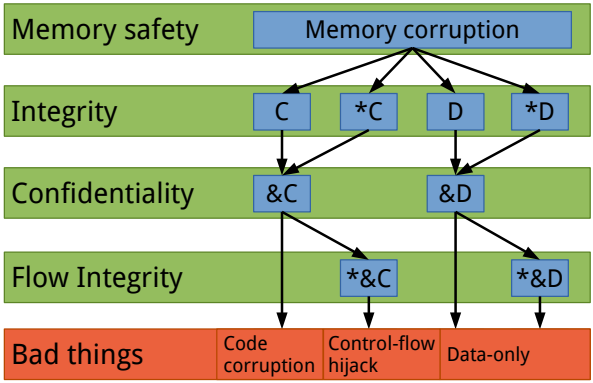


Figure 5.2: The attack flow of memory corruption-based attacks. 'C' conforms to Code, 'D' to Data; '&' marks an address-of operator; '*' marks a dereference operator. The attack path needs to bypass defenses at different levels of abstraction: integrity, confidentiality, and flow integrity.

5.4.1 Control-Flow Hijacking

A successful *Control-Flow Hijacking* attack redirects the application's control-flow to an adversary-controlled location. This attack primitive gives the adversary control over the execution of the program and instruction pointer. Control-flow hijacking is generally achieved by overwriting a code pointer either directly or indirectly. An indirect control-flow transfer such as an indirect call (`call rax` for x86), indirect jump (`jmp rax` for x86), or indirect branch (`mov pc, r2` for ARM) updates the instruction pointer with a value from a register, continuing control-flow from this new location. The value is often read from writable memory and can therefore be controlled by the attacker.

Given code integrity (i.e., benign code cannot be modified), the attacker cannot modify relative control-flow transfers. A direct control-flow transfer is a call or jump to a hard coded location where the target is either in read-only memory or encoded as part of the instruction. For example on x86, function calls are encoded as control-flow transfers relative to the instruction pointers with an offset that is part of the instruction itself.

Indirect calls are used to provide flexibility as part of the programming language, e.g., a call through a function pointer in C or a virtual dispatch for C++. Additionally, the return instruction pointer on the stack points back to the location in the calling function. At the programming language level, the target is well defined and given type safety and memory safety, the control-flow of the program is well contained. But given any memory safety or type safety corruption, the value of the

target in memory may be overwritten by the attacker.

```
1  int benign();
2
3  void vuln(char *attacker) {
4    int (*func)();
5    char buf[16];
6
7    // Function pointer is set to benign function
8    func = &benign;
9
10   // Buffer overflow may compromise memory safety
11   strcpy(buf, attacker);
12
13   // Attacker may hijack control-flow here.
14   func();
15 }
```

Listing 5.1: Buffer overflow into function pointer.

In the code example above, an attacker may supply a buffer that is larger than the local stack buffer. When copying data into the buffer `buf`, a stack-based buffer overflow may overwrite the function pointer on the stack. This memory safety violation allows the adversary to compromise the stored code pointer `func`. When `func` is later dereferenced, the attacker-controlled value is used instead of the original value.

To orchestrate a control-flow hijack attack, an adversary must know the location of the code pointer that will be overwritten and the location of the target address, i.e., where to redirect

control-flow to. Additionally, there must be a path from the vulnerability to the location where the attacker-controlled code pointer is dereferenced.

5.4.2 Code Injection

Code injection assumes that the attacker can write to a location in the process that is executable. The attacker can either overwrite or modify existing code in the process, rewriting instructions either partially or completely. Corrupting existing code may allow an attacker to gain code execution without hijacking control flow as a benign control-flow transfer may direct execution to attacker-controlled code. Alternatively, the attacker can hijack control-flow to the injected code.

Modern architectures support the separation of code and data, therefore this attack vector is no longer as prevalent as it was. A modern variant of this attack vector targets Just-In-Time compilers that generate new code dynamically. In such environments some pages may be writable and executable at the same time.

Code injection requires the existence of a writable and executable memory area, the knowledge of the location of this memory area, and, if the location is not reached through a benign control-flow transfer, a control-flow hijack primitive. The injected code conforms to shellcode that, depending on the vulnerability, must follow certain guidelines. See Section 8.1 for a discussion of shellcode and its construction.

Given the vulnerable code in the example below, an attacker can provide input to overflow the `cookie` buffer, continuously overwriting information that is higher up on the stack as the `strcpy` function does not check the bounds of `cookie`. Assume that this program is compiled without code execution prevention and the program runs without ASLR.

There are several methods to exploit this vulnerability. Control-flow hijacking is achieved by overwriting the return instruction pointer on the stack. Code may be injected in three locations: (i) the buffer itself, (ii) higher up on the stack frame "above" the return instruction pointer, or (iii) in an environment variable (the example conveniently reports the location of the `EGG` environment variable. We therefore prepare the shellcode in the environment variable `EGG` and overwrite the return instruction pointer to point to the beginning of `EGG`. Upon return, instead of returning control flow to the calling function, the shellcode in `EGG` will be executed, giving the attacker control. The full input to exploit the vulnerability is: `[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]`, [the address of the `EGG` environment variable], and a `[0x0]` byte. The first 32 bytes fill the buffer with `A`s, the next 4 bytes overwrite some padding inserted by the compiler, the next 4 bytes overwrite the frame pointer, and the remaining 4 bytes overwrite the return instruction pointer. The exact layout of the stack frame depends on the compiler and may be inferred by analyzing the assembly code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char cookie[32];
  printf("Give me a cookie (%p, %p)\n",
    cookie, getenv("EGG"));
  strcpy(cookie, argv[1]);
  printf("Thanks for the %s\n", cookie);
  return 0;
}
```

Listing 5.2: Stack based code injection.

5.4.3 Code Reuse

Instead of injecting code, *reuse* existing code of the program. The main idea is to stitch together existing code snippets to execute new arbitrary behavior. This is also called Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP), Call-Oriented Programming (COP), Counterfeit-Object Oriented Programming (COOP) for different aspects of code reuse.

Any executable code of the application may be used by the attacker in an alternate way under the constraints of any active mitigation mechanism. Through indirect control-flow transfers, adversaries can chain together small code sequences (called gadgets) that end in another indirect control-flow transfer.

A successful code reuse attack requires (i) knowledge of a writable memory area that contains *invocation frames* (gadget address and state such as register values), (ii) knowledge of executable code snippets (*gadgets*), (iii) control-flow must be hijacked/redirected to prepared invocation frames, and (iv) construction of ROP payload. See Section 8.2 for a discussion on the construction of ROP payloads.

5.5 Summary

This chapter presented a broad overview of different attack vectors. We discussed attack vectors from a simple Denial of Service (DoS) that prohibits a legit user from using a service over information leakage that allows an adversary to extract secrets to privilege escalation and confused deputies which both give the adversary additional computation privileges that they would otherwise not have.

The goals of adversaries vary and weaknesses in software may be used to achieve different goals. The craft of an attacker is to leverage an exposed bug to trick the underlying program into doing something on the attacker's behalf.

6 Defense Strategies

Defending against software vulnerabilities is possible along four dimensions: (i) formally proving software correct which guarantees that the code is bug free (according to a given specification), (ii) rewriting the software in a safe programming language, (iii) software testing which discovers software flaws before they can do any harm, and (iv) mitigations which protect a system in the presence of unpatched or unknown vulnerabilities.

6.1 Software Verification

Software verification proves the correctness of code according to a given specification. The security constraints (e.g., no memory or type safety violation) are encoded and given as configuration to the verification process. Different forms of formal verification exist such as bounded model checking or abstract interpretation. All of them prove that a given piece of code conforms to the formal specification and guarantee that no violations of the security policy are possible.

Some well-known examples of formally verified code are seL4 [14], a formally verified operating system kernel or CompCert

[19], a formally verified compiler. For seL4, operating system concepts are encoded as high-level policies on top of a proof system. After proving the correctness of the high level operating system policy, the equivalence between the high-level policy and a low-level implementation is proven in a second step. For CompCert, individual steps and transformations of the compiler are provably correct. This guarantees that verified compiler transformations are always correct and will not introduce any bugs as part of the compilation process.

The main challenge of software verification is scalability. Automatic software verification scales to 100s of lines of code with an exponential increase in verification cost with linear increase of code. As an example, human guided verification of the seL4 kernel verification cost several person years.

6.2 Language-based Security

Language-based security is a rather new area of research that focuses on enforcing security properties as part of the programming language, protecting the programmer from making mistakes. Programming languages have always enforced some form of structure, thereby protecting against certain types of bugs. Language-based security makes reasoning about security aspects explicit and allows languages to be designed in a security specific way.

Early examples are functional programming languages that inherently protect against memory safety violations and data

races. Functional languages track references to data and prohibit direct pointer manipulation. Orthogonally, variables may only be written once during assignment, protecting against data races as data cannot be modified concurrently. Java is a popular programming language that enforces both type safety and memory safety as part of the programming language and its runtime system.

A modern example of a *secure* imperative programming language is Rust. Rust enforces memory safety and data race freedom through ownership tracking and a strict type system. The clear ownership in Rust prohibits concurrent modification and allows the compiler to check memory ownership during compilation. The majority of the type checks are executed statically as part of the compilation (and the compiler can give detailed warnings about possible issues) with minimal runtime checks. Rust gives strong memory safety, type safety, and data race freedom guarantees at negligible performance overhead at the cost of a steep learning curve as the programmer must make all ownership assumptions explicit when writing code.

6.3 Testing

Software testing allows developers to identify bugs before they can do any harm. This process is orthogonal to software development and, if done correctly, is integrated into the development process to allow for continuous software testing. Testing is the process of *executing a program* to *find flaws*. An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification of functional

requirements (features a, b, c) or operational requirements (performance, usability). Both functional and operational requirements are testable. Security requirements are not directly testable as, e.g., the absence of bugs is hard to prove.

For applications written in C/C++ we can indirectly test that memory safety and type safety guarantees hold by observing the effects of testing. Instead of checking the correct computation of the result we measure if the program crashes or is terminated through a security exception.

> Testing can only show the presence of bugs, never their absence.
> (Edsger W. Dijkstra)

6.3.1 Manual Testing

Test-driven development flips the process between testing and implementation. Instead of writing test cases for a given implementation, the test cases are written as a first implementation step. This approach allows the programmer to encode details that are specified in the design. The test cases are witnesses for required features. Initially all test cases fail and slowly, as the implementation proceeds, the test cases start to pass as more and more features are implemented. Testing should be integrated with a *continuous integration system* that verifies all test cases (and coding style) whenever new code is checked into the project's source repository.

Manual testing involves the development of positive and negative test cases and embedding assertions in the production

code. Assertions help test negative test cases and find bugs before they corrupt any state which would make them hard to triage.

Unit tests are small test cases that focus on an individual unit or feature of the program. The Google testing framework [7] simplifies the implementation of such tests. To provide necessary application state, unit testing frameworks enable a wide range of mock operations.

Integration tests allow testing of interactions between individual modules. For example, an integration test could measure the interaction between a browser's DOM and the printer daemon that creates a visual representation of the DOM to send it off to a printer.

System testing tests the full application. For example, a browser displaying a web page and a user interacting with that page to accomplish a certain task such as filling out a form and sending it off.

Beta testing leverages a small set of users that thoroughly test the software to find remaining flaws. Any identified bug is triaged and fixed. It is good testing practice to create a test case for each identified flaw to protect against regression. Regression happens if a code change suddenly fails existing test cases. Keeping test cases for each fixed bug allows early detection if a bug is introduced again and may catch similar bugs as well.

An interesting question is what metric is used to evaluate the quality of a test suite. A deterministic metric allows an absolute evaluation of the quality of the suite, i.e., how well

a test suite maps to a program. Coverage is a natural metric that suits the aforementioned criteria. Coverage is used as a metric to evaluate the quality of a test suite. The intuition is that a software flaw is only detected if the flawed code is executed. The effectiveness of the test suite therefore depends on the resulting coverage. Different coverage metrics exist with varying tradeoffs. We consider statement coverage, branch coverage, path coverage, and data-flow coverage.

Statement coverage measures, for each statement, if it has been executed. Coverage tracking can be done using a simple array and instrumentation that marks the executed bit for each statement when executed (or basic block without loss of generality). A disadvantage of statement coverage is that not all edges are tracked, e.g., the backward edge of a loop may never be executed in the following example:

```
1 int func(int elem, int *inp, int len) {
2   int ret = -1;
3   for (int i = 0; i <= len; ++i) {
4     if (inp[i] == elem) { ret = i; break; }
5   }
6   return ret;
7 }
```

Listing 6.1: Example where statement coverage misses a bug.

The test input `elem = 2, inp = [1, 2], len = 2` achieves full statement coverage but the execution never executed the last iteration of the loop which will result in a buffer overflow.

The branch edge from the check in the loop to the end of the loop is never followed.

Branch coverage measures, for each branch, if it has been followed. Again, coverage tracking can be done using a simple array and instrumentation that marks executed branches. Branch coverage marks both the execution of the basic block *and* the branch to a given basic block and is therefore a super set of simple statement coverage. Full branch coverage implies full statement coverage. Unfortunately, branch coverage may not be precise enough:

```
1 int arr[5] = { 0, 1, 2, 3, 4 };
2 int func(int a, int b) {
3   int idx = 4;
4   if (a < 5) idx -= 4; else idx -= 1;
5   if (b < 5) idx -= 1; else idx += 1;
6   return arr[idx];
7 }
```

Listing 6.2: Limitation of branch coverage.

The test inputs `a = 5, b = 1` and `a = 1, b = 5` achieve full branch coverage (and full statement coverage), yet, not all possible paths through the program will be executed. The input `a = 1, b = 1` results in a bug when both statements are true at the same time. Full path coverage evaluates all possible paths. Evaluating all possible paths quickly becomes expensive as each branch doubles the number of evaluated paths or even impossible for loops where the bounds are not known. This exponential increase in the amount of paths is

called *path explosion*. Loop coverage (execute each loop `0`, `1`, `n` times), combined with branch coverage probabilistically covers state space. Implementing path coverage requires runtime tracing of the paths as, programs with more than roughly 40 branches cannot be mapped into a flat array and enumerating all paths becomes impossible given current (and future) memory constraints.

Data-flow coverage extends beyond path coverage and tracks full data flow through the program at even higher overhead. In addition to path constraints (a boolean for each path decision), the values of all program values have to be tracked as well. This is the most precise way to track coverage but involves high overheads.

Therefore, in practice, branch coverage is the most efficient tool and several mechanisms exist that allow branch coverage tracking for software. Two examples are gcov and SanitizerCoverage. Branch coverage keeps track of edges in the CFG, marking each executed edge with the advantage that only a bit of information is required for each edge (and no dynamic information that depends on the number of executed paths).

6.3.2 Sanitizers

Test cases detect bugs through miscompared results, assertion failures, segmentation faults, division by zero, uncaught exceptions, or mitigations that trigger process termination. *Sanitizers* are compilation frameworks that instrument a program with additional checks. When executed with the test

cases, unit tests, or under fuzz testing, the sanitizers can detect violations at the source of the flaw and not just when the process traps. Sanitizers detect low level violations of, e.g., memory safety or type safety, not high-level functional properties.

Recently several new sanitizers were added to the LLVM compiler framework to target different kinds of vulnerabilities: AddressSanitizer, LeakSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer, ThreadSanitizer, and HexType.

AddressSanitizer (ASan) [30] detects memory errors. It places red zones around objects and checks those objects on trigger events. The typical slowdown introduced by ASan is 2x. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Note that the ASan memory safety guarantees are probabilistic. ASan leverages so called red zones around objects which are marked with a special value. Checks ensure that the special value remains intact. Similarly for use-after-free, instrumentation ensures that the values remain correct. This obviously does not protect against memory areas that are reallocated to different objects. ASan is therefore not a mitigation but a sanitizer that helps to probabilistically detect flaws.

LeakSanitizer detects run-time memory leaks. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode. LSan adds almost no performance overhead until process termination, when the extra leak detection phase runs.

MemorySanitizer (MSan) detects uninitialized reads. MSan uses heavy-weight program transformation to keep state of allocated objects. Memory allocations are tagged and uninitialized reads are flagged. The typical slowdown of MSan is 3x. Note: do not confuse MemorySanitizer (detects uninitialized reads) and AddressSanitizer (detects spatial memory safety violations and probabilistic memory reuse).

UndefinedBehaviorSanitizer (UBSan) detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs. Slowdown depends on the amount and frequency of checks. This is the only sanitizer that can be used in production. For production use, a special minimal runtime library is used with minimal attack surface. Detectable errors are:

- Unsigned/misaligned pointers
- Signed integer overflow
- Conversion between floating point types leading to overflow
- Illegal use of NULL pointers
- Illegal pointer arithmetic
- and many more (check the documentation)

ThreadSanitizer (TSan) detects data races between threads. It instruments writes to global and heap variables and records

which thread wrote the value last, allowing detecting of Write-After-Write, Read-After-Write, Write-After-Read data races. The typical slowdown of TSan is 5-15x with 5-15x memory overhead.

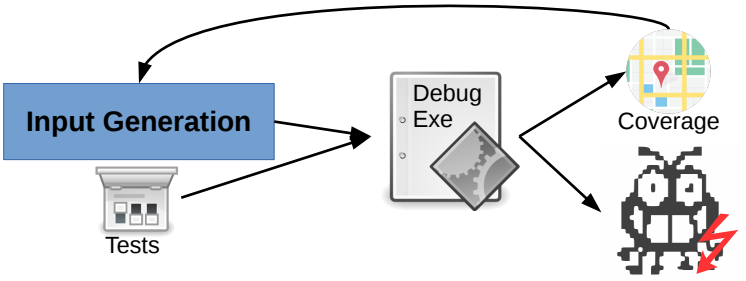
HexType [11] detects type safety violations (type confusion). It records the true type of allocated objects and makes all type casts explicit. HexType implements type safety for C++. The typical slowdown of HexType is 1.5x.

Alternatively, Valgrind [24] implements a sanitization framework for binaries. Binaries are lifted into a high-level representation that is instrumented. During execution, metadata is kept depending on the selected instrumentation of the sanitizer. Valgrind implements different memory safety and thread safety sanitizers.

6.3.3 Fuzzing

Dynamic analysis techniques leverage a concrete execution through the program to test a given policy. *Fuzz testing* is a simple approach that creates program input to generate different traces through the program with the intention to trigger a crash. Crashing inputs are then collected and triaged to fix bugs. Different approaches for fuzz testing exist with different levels of program cooperation. Fuzzing can leverage information about the input structure or the program structure to improve over blind random input mutation.

While fuzzing, the process of providing random input to a program to trigger unintended crashes, has been around for



Overview of the fuzzing process.

decades, we have recently seen a revival of techniques with several papers improving fuzzing effectiveness at each top tier security conference. The idea behind fuzzing is incredibly simple: execute a program in a test environment with random input and detect if it crashes. The fuzzing process is inherently sound but incomplete. By producing test cases and observing if the program under test crashes, fuzzing produces a witness for each discovered crash. As a dynamic testing technique, fuzzing is incomplete as it will likely neither cover all possible program paths nor data-flow paths except when run for an infinite amount of time. Fuzzing has seen a massive amount of attention in recent years both from industry where fuzzing is used to discover bugs to academia where new fuzzing techniques are developed. Fuzzing strategies are inherently an optimization problem where the available resources are used to discover as many bugs as possible, covering as much of the program functionality as possible through a probabilistic exploration process. Due to its nature as a dynamic testing technique, fuzzing faces several unique challenges:

- Input generation: fuzzers generate inputs based on a mutation strategy to explore new state. The underlying strategy determines how effectively the fuzzer explores a given state space. A challenge for input generation is the balance between exploring new control flow and data flow.
- Detecting flaws: to discover flaws, fuzzers must distinguish between benign and buggy executions. Not every bug results in an immediate segmentation fault and detecting state violation is a challenging task, especially as code generally does not come with a formal model.
- Preparing programs: fuzzing struggles with some aspects of code such as fuzzing a complex API, checksums in file formats, or hard comparisons such as password checks. Preparing the fuzzing environment is a crucial step to increase the efficiency of fuzzing.
- Evaluating fuzzing effectiveness: defining metrics to evaluate the effectiveness for a fuzzing campaign is challenging. For most programs the state space is (close to) infinite and fuzzing is a brute force search in this state space. Deciding when to, e.g., move to another target, path, or input is a crucial aspect of fuzzing. Comparing different fuzzing techniques requires understanding of the strengths of a fuzzer and the underlying statistics to enable fair comparison.

AFL [36] is the state-of-the art fuzzer that uses mutational input generation. AFL uses grey-box instrumentation to track branch coverage and mutate fuzzing seeds based on previous

branch coverage. Branch coverage tracks the last two executed basic blocks (resulting in a crude approximation of path coverage). New coverage is detected on the history of the last two branches.

6.3.3.1 Input generation

Input generation is the first of two essential parts of the fuzzing process. Every fuzzer must automatically generate test cases to be run on the execution engine. The cost for generating a single input should be low, following the underlying philosophy of fuzzing where iterations are cheap. There are two fundamental forms of input generation: *model-based input generation* and *mutation-based input generation*. The first is aware of the input format while the latter is not.

Knowledge of the input structure given through a grammar enables *model-based input generation* to produce (mostly) valid test cases. The grammar specifies the input format and implicitly the explorable state space. Based on the input specification, the fuzzer can produce valid test cases that satisfy many checks in the program such as valid state checks, dependencies between fields, or checksums such as a CRC32. For example, without an input specification the majority of randomly generated test cases will fail the check for a correct checksum and quickly error out without triggering any complex behavior. The input specification allows input generation to balance the generated test inputs according to the underlying input grammar. The disadvantage of grammar-based input generation is the need for a concrete input specification. Most input formats are not

formally described and will require an analyst to define the intricate dependencies.

Mutation-based input generation requires a set of seed inputs that trigger valid functionality in the program and then leverages random mutation to modify these seeds. Providing a set of valid inputs is significantly easier than formally specifying an input format. The input mutation process then constantly modifies these input seeds to trigger interesting behavior.

Orthogonally to the awareness of the input format, a fuzzer can be aware of the program structure. Whitebox fuzzing infers knowledge of the program structure through program analysis or relies on an analyst to custom-tailor fuzzing for each tested program, resulting in untenable cost. Blackbox fuzzing blindly generates new input without reflection, severely limiting progress of the fuzzer. Greybox fuzzing leverages program instrumentation instead of program analysis to infer coverage during the fuzzing campaign itself, merging analysis and testing.

Coverage-guided greybox fuzzing combines mutation-based input generation with program instrumentation to detect whenever a mutated input reaches new coverage. Program instrumentation tracks which areas of the code are executed and the coverage profile is tied to specific inputs. Whenever an input mutation generates new coverage, it is added to the set of inputs for mutation. This approach is incredibly efficient due to the low cost instrumentation but still results in broad program coverage.

Modern fuzzing is heavily optimized and focuses on efficiency,

measured by the number of bugs found per time. Sometimes, fuzzing efficiency is generalized as the number of crashes found per time, but this may lead to problems as crashes may not be unique and many crashes point to the same bug.

6.3.3.2 Execution engine

After generating the test cases, they must be executed in a controlled environment to observe when a bug is triggered. The execution engine takes the produced input, executes the program under test, extracts runtime information such as coverage, and detects crashes. Ideally a program would terminate whenever a flaw is triggered. For example, an illegal pointer dereference on an unmapped memory page results in a segmentation fault which terminates the program, allowing the executing engine to detect the flaw. Unfortunately, only a small subset of security violations will result in program crashes. Buffer overflows into adjacent memory locations for example, may only be detected later if the overwritten data should be used or may never be detected at all. The challenge for this component of the fuzzing process is to efficiently enable the detection of policy violations. For example, without instrumentation only illegal pointer dereferences to unmapped memory, control-flow transfers to non-executable memory, division by zero, or similar exceptions will trigger a fault.

To make security policies tractable, the program under test may be instrumented with additional checks that detect violations early. Safety violations through undefined behavior for code written in systems languages are particularly tricky.

Sanitization analyzes and instruments the program during the compilation process to enforce selected properties. Address Sanitizer [30], the most commonly used sanitizer, probabilistically detects spatial and temporal memory safety violations by placing red-zones around allocated memory objects, keeping track of allocated memory, and carefully checking memory accesses. Other sanitizers cover undefined behavior, uninitialized memory, or type safety violations [11]. Each sanitizer requires certain instrumentation that increases the performance cost. The usability of sanitizers for fuzzing therefore has to be carefully evaluated as, on one hand, it makes error detection more likely but, on the other hand, reduces fuzzing throughput.

6.3.3.3 Preparing programs

The key advantage of fuzzing is its incredible simplicity (and massive parallelism). Due to this simplicity, fuzzing can get stuck in local minima where continuous input generation will not result in additional crashes or new coverage – the fuzzer is stuck in front of a coverage wall. A common approach to circumvent the coverage wall is to extract seed values used for comparisons. These seed values are then used during the input generation process. Orthogonally, a developer can comment out hard checks such as CRC comparisons or checks for magic values. Removing these non-critical checks from the program requires that the developer is aware of what are critical safety checks and what can be safely commented out.

Several recent extensions [31:@sanjay17ndss, @peng18sp, @insu18sec] try to bypass the coverage wall by detecting when

the fuzzer gets stuck and then leveraging an auxiliary analysis to either produce new inputs or to modify the program. It is essential that this (sometimes heavy-weight) analysis is only executed infrequently as alternating between analysis and fuzzing is costly and reduces fuzzing throughput.

The concept of fuzzing libraries also faces the challenge of experiencing low coverage during unguided fuzzing campaigns. Programs often call exported library functions in sequence, building up complex state in the process. The library functions execute sanity checks and quickly detect illegal or missing state. These checks make library fuzzing challenging as the fuzzer is not aware of the dependencies between library functions. Existing approaches such as libFuzzer [32]] require an analyst to prepare a test program that calls the library functions in a valid sequence to build up the necessary state to fuzz complex functions.

6.3.3.4 Evaluating fuzzing

At a high-level, evaluating fuzzing is straightforward: if technique A finds more bugs than technique B, then technique A is superior to technique B. In practice, there are challenging questions that must be answered such as for how long the techniques are evaluated, how bugs are identified, or what the fuzzing environment is. A recent study [13] evaluated the common practices of recently published fuzzing techniques (and therefore also serves as overview of the current state of the art). The study identified common benchmarking crimes and condensed their findings into five recommendations:

- A single execution is not enough due to the randomness in the fuzzing process. To evaluate different mechanisms, we require multiple trials and statistical tests to measure noise.
- A single target is not enough to evaluate a fuzzer. Instead, fuzzers should be evaluated across a broad set of target programs to highlight advantages and disadvantages of a given configuration.
- Heuristics cannot be used as the only way to measure performance. For example, collecting crashing inputs or even stack bucketing does not uniquely identify bugs. Ground truth is needed to disambiguate crashing inputs and to correctly count the number of discovered bugs. A benchmark suite with ground truth will help.
- The choice of seeds must be documented as different seeds provide vastly different starting configurations and not all techniques cope with different seed characteristics equally well.
- Fuzzing campaigns are generally executed for multiple days to weeks. Comparing different mechanisms based on a few hours of execution time is not enough. Fuzzing must be evaluated for at least 24 hours, maybe even longer.

6.3.3.5 Future fuzzing work

Fuzzing is currently an extremely hot research area in software security with several new techniques being presented at each

top tier security conference. The research directions can be grouped into improving input generation, reducing the performance impact for each execution, better detection of security violations, or pushing fuzzing to new domains such as kernel fuzzing or hardware fuzzing. All these areas are exciting new dimensions and it will be interesting to see how fuzzing can be improved further.

6.3.3.6 Fuzzing Summary

With the advent of coverage-guided greybox fuzzing, dynamic testing has seen a renaissance with many new techniques that improve security testing. While incomplete, the advantage of fuzzing is that each reported bug comes with a witness that allows the deterministic reproduction of the bug. Sanitization, the process of instrumenting code with additional software guards helps to discover bugs closer to their source. Overall, security testing remains challenging, especially for libraries or complex code such as kernels or large software systems. Given the massive recent improvements of fuzzing, there will be exciting new results in the future. Fuzzing will help make our systems more secure by finding bugs during the development of code before they can cause any harm during deployment.

6.3.4 Symbolic Execution

Static analysis techniques analyze the source code (or binary) for violations of a given policy. Static analysis frameworks usually combine a wide set of techniques to discover different types

of vulnerabilities based on abstract interpretation of the code combined with control-flow and data-flow analysis. Abstract interpretation transforms the semantics of the programming language to simplify source code, translating it into a form that allows reasoning over an abstract grammar. The advantage of static analysis techniques is their ubiquity and simple application to large code bases. A disadvantage is that they may lead to large amounts of false positives.

Symbolic execution is an analysis technique that is somewhat between static and dynamic analysis with many different flavors. A symbolic execution engine reasons about program behavior through "execution" with symbolic values. Concrete values (input) are replaced with symbolic values. Symbolic values can have any value, i.e, variable x instead of value `0x15`. Symbolic values capture all possible values. The symbolic state of the application is tracked through a set of collected constraints. Operations (read, write, arithmetic) become constraint collection/modification operations as they add new constraints to the collection, possibly summarizing existing constraints. Symbolic execution allows *unknown* symbolic variables in the evaluation.

Through this abstract interpretation, the program is turned into a set of constraints. Instead of executing the program with concrete input, all memory becomes symbolic and computation updates the symbolic values. This allows a large amount of traces through the program to be evaluated at the same time. Symbolic execution is limited through the complexity of the constraints. For each branch in the program, the amount of state doubles as either the true or the false branch can be taken

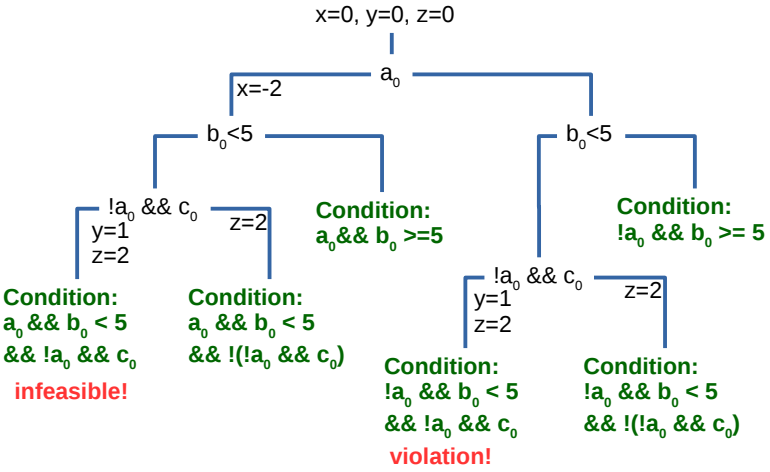
which leads to a state explosion.

```
1 void func(int a, int b, int c) {
2   int x = 0, y = 0, z = 0;
3   if (a) x = -2;
4   if (b < 5) {
5     if (!a && c) y = 1;
6     z = 2;
7   }
8   assert(x + y + z != 3);
9 }
```

Listing 6.3: Symbolic execution example. The parameters a, b, and c are symbolic.

A path condition is a quantifier-free formula over symbolic inputs that encodes all branch decisions (so far). To determine whether a path is feasible, the symbolic execution engine checks if the path condition is satisfiable. Given the set of constraints, an SMT solver provides satisfying assignment, counter example, or timeout.

While symbolic execution gives a precise evaluation of all paths in the program, it has a hard time with loops and recursions which result in infinite execution traces. Path explosion is another challenge as each branch doubles the number of paths and state that is tracked. Environment modeling, e.g., through system calls is also complex due to the amount of operating system state that must be modeled. Lastly, symbolic data where both the array data and the index are symbolic is challenging as arbitrary data increases the number of possible solutions.



Constraint tracking along the different paths for the symbolic execution example.

All these problems have in common that the complexity makes the constraints explode, reducing the chances that the SMT solver will find a solution before a timeout.

Concolic testing addresses the problems of symbolic execution by leveraging a concrete execution trace to "base" the symbolic execution to places nearby. Only constraints close to and along the recorded trace are evaluated.

KLEE [6] is an example of a symbolic/concolic execution engine based on the LLVM compiler. LLVM compiles the target program with instrumentation for symbolic/concolic execution. KLEE then models the environment and provides a selection of many different search strategies and heuristics to constrain symbolic execution.

6.4 Mitigations

Mitigations are the last line of defense against software flaws that violate low level security policies such as memory safety, type safety, or integer overflows. Logic flaws are out of scope for mitigations as they are dependent on the requirements and specification of an application which is (generally) not expressed in a machine-readable way. Given that code was neither verified nor tested for a bug, mitigations can check for policy violations at runtime. Mitigations against flaws generally result in some performance overhead due to these additional checks. The majority of mitigations are therefore designed to incur negligible performance or memory overhead, at the trade-off of lower security guarantees. The reason why overhead is

not tolerated is the abstract risk of bugs. Mitigations protect against unpatched and unknown bugs and, therefore, against an abstract risk. The cost of running the mitigation is real.

The set of deployed mitigations is Data Execution Prevention (DEP) to protect against code injection, Address Space Layout Randomization (ASLR) to probabilistically protect against information leaks, stack canaries to protect backward edge control-flow, safe exception handling to protect against injected C++ exception frames, and fortify source to protect against format string attacks. Some stronger mitigations such as Control-Flow Integrity (CFI), sandboxing, stack integrity, and software-based fault isolation are being deployed on highly exposed software with broad dissemination likely happening soon.

6.4.1 Data Execution Prevention (DEP)/W^X

Most widespread hardware did initially not distinguish between code and data. Any readable data in a process' address space could be executed by simply transferring control-flow to that data location. The memory management unit allowes pages to be unmapped, readable, or writable. These three different configurations are handled through a single bit in the page table that marks if a page is writable or only readable. If a page is not mapped then it is neither writable nor readable. Any mapped page is always readable.

Data Execution Prevention (DEP) or W^X (writable xor executable) enforces that any location in memory is either executable or writable but never both. DEP enforces code integrity,

i.e., code cannot be modified or injected by an adversary. In the absence of a just-in-time compiler or self-modifying code in the process, code remains static and limited to the initial set of executable code as loaded when the process started.

The assumption for designing this mitigation was that enabling the CPU to distinguish between code and data would stop code execution attacks. Modern architectures extended the page table layout with an additional No-eXecute bit (Intel calls this bit eXecute Disable, AMD calls it Enhanced Virus Protection, and ARM calls it eXecute Never). This bit allows the MMU to decide, on a per-page basis, if it is executable or not. If the bit is set, then data on that page cannot be interpreted as code and the processor will trap if control flow reaches that page. Not all hardware supports the necessary extensions and several software-only extensions were designed to give similar guarantees, often at higher performance cost. Figure 6.1 shows the changes to a process' address space under DEP/W^X.

This mitigation is a prime example of a successful mitigation that results in negligible overhead due to a hardware extension. The hardware-enabled mitigation is now used generally and widely. DEP and W^X stop an attacker from injecting new executable code in the address space of the application. However, without any other mitigation, an application is still prone to code reuse. See Section 5.4.2 and 5.4.3 for more details.

6.4.2 Address Space Layout Randomization (ASLR)

Any successful control-flow hijack attack depends on the attacker overwriting a code pointer with a known alternate target.

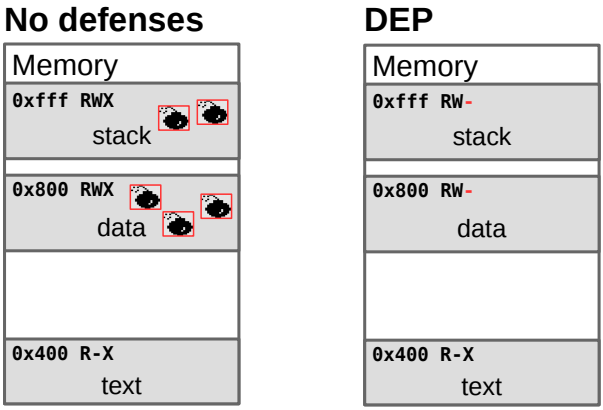


Figure 6.1: Changes to the address space of a process for DEP/W^X. Bombs show memory areas an exploit may modify.

Address space randomization changes (randomizes) the process memory layout. If the attacker does not know where a piece of code (or data) is, then it cannot be reused in an attack. Under address space randomization, an attacker must first *learn* and *recover* the address layout. Alternatively, an attacker may cleverly reuse existing pointers at well-known relative offsets.

Challenges for address space randomization are information leakage through side channels or other leaks, low entropy that enables brute forcing of the relocation strategy, and rerandomization as long running processes will have their layout fixed after the start of the process (due to performance trade-offs as rerandomization would be costly). The security improvement of address space randomization depends on (i) the entropy available for each randomized location, (ii) the completeness of

randomization (i.e., are all objects randomized), and (iii) the lack of any information leaks.

Address space randomization features several candidates that can be placed at random locations in the address space:

- Randomize start of heap;
- Randomize start of stack;
- Randomize start of code (PIE for executable, PIC for libraries);
- Randomize code at the instruction level (resulting in prohibitive overhead);
- Randomize mmap allocated regions;
- Randomize individual allocations (malloc);
- Randomize the code itself, e.g., gap between functions, order of functions, basic blocks;
- Randomize members of structs, e.g., padding, order.

There are different forms of fine-grained randomization with different performance, complexity, and security trade-offs. Address Space Layout Randomization (ASLR) is a form of address space randomization that leverages virtual memory to randomize parts of an address space. ASLR shuffles the *start addresses of* the heap, the stack, all libraries, the executable, and mmapped regions. ASLR is inherently page based (to limit overhead) and the main cost is due to position independent code [26].

ASLR requires virtual memory and support from the operating system, linker/loader, and compiler. The implementation of ASLR is straightforward and fits well into the virtual address space provided by operating systems. When loading a library,

allocating a stack, or mmapping a region it has to be placed somewhere in memory. Randomizing the low bits of the page base address implements address space randomization at the page level. Virtual memory is required to allow reshuffling of addresses. The operating system must allow randomization for random address for the initial binary and mmapped regions. The linker/loader must prepare the process at a random location in memory. The compiler must ensure that code references other code locations relatively as each code block may be at a different location every time the process starts. Figure 6.2 shows the change to the address space under ASLR.

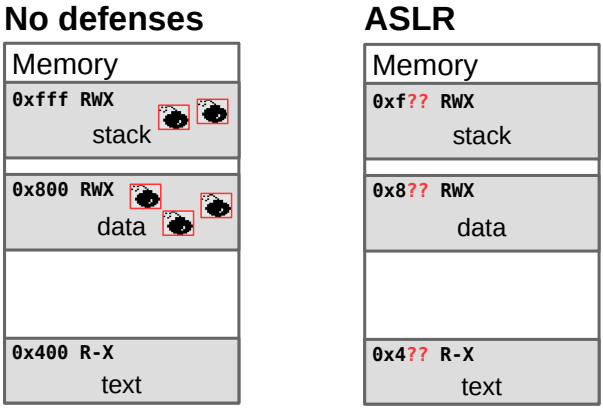


Figure 6.2: Changes to the address space of a process under ASLR.

The entropy of each section is key to security (if all sections are randomized). For example, Figure 6.2 uses 8 bit of entropy for each section. An attacker follows path of least resistance, i.e., targets the object with the lowest entropy. Early ASLR

implementations had low entropy on the stack and no entropy on x86 for the main executable (non-PIE executables). Linux (through Exec Shield) uses 19 bits of entropy for the stack (16 byte aligned) and 8 bits of mmap entropy (4096 byte/page aligned).

6.4.3 Stack integrity

Early code execution attacks often targeted stack-based buffer overflows to inject code. An early defense targeted precisely these buffer overflows. While memory safety would mitigate this problem, adding full safety checks is not feasible due to high performance overhead. Instead of checking each dereference to detect arbitrary buffer overflows we can add a check for the integrity of a certain variable. The goal for this mitigation is to protect an application against stack-based overflows that change the stored return instruction pointer or saved stack base pointer. Stack integrity as a property ensures that both the return instruction pointer and the stack pointer cannot be modified illegally.

Legal modifications of the return instruction pointers include, e.g., a trampoline that redirects control flow to an alternate location (overwriting the stored return instruction pointer from a call instruction) or so-called thunks that pop the return instruction pointer into a general purpose register (allowing code to infer the current value of the instruction pointer through a `call nextInstruction; pop generalPurposeRegister` sequence for ISAs such as x86 that do not allow explicit access to the instruction pointer). A stack pivot changes the stack pointer

and shifts the current stack frame to an alternate location under the attacker's control, e.g., by overwriting a spilled stack pointer value.

Different mitigations implement some form of stack integrity. The most common forms are stack canaries that place guards around sensitive values, shadow stacks that store a copy of sensitive data in an alternate location, and safe stacks which split the stacks into sensitive, protected data and unprotected data.

Interesting challenges for mitigations are, next to security guarantees, the support for exceptions in C++, setjmp/longjmp for C programs, and tail call optimizations which replace the call to a leaf function to reuse the same stack frame. These features allow abnormal control flow on the return edge and transfer code to stack frames higher up on the stack, potentially skipping several frames in the chain.

6.4.3.1 Stack canaries

The key insight for stack canaries is that, in order to overwrite the return instruction pointer or base stack pointer, all other data on the way to those pointers must be overwritten as well. This mitigation places a canary before the critical data and adds instrumentation to (i) store the canary when the function is entered and (ii) check its integrity right before the function returns. The compiler may place all buffers at the end of the stack frame and the canary just before the first buffer. This way, all non-buffer local variables are protected against sequential

overwrites as well. Stack canaries are a purely compiler-based defense.

The term stack canaries comes from mine workers who brought canary birds along when they went into coal mines. The birds would pass out from the lack of oxygen and alert the mine workers, similar to how stack canaries signal a buffer overflow when data adjacent to the return instruction pointer is overwritten.

Figure 6.3: Canary bird, used by rescue workers in coal mines. From WATFORD, 1970, public domain.

The weakness of this defense is that the stack canary only protects against continuous overwrites as long as the attacker does not know the canary. If the attacker knows the secret or the attacker uses a direct overwrite then this mitigation is

not effective. An alternative to protect the return instruction pointer through a canary is to encrypt the return instruction pointer, e.g., by xoring it with a secret. The stack canary instrumentation is surprisingly simple (note that, to support a per-thread unique canary, this implementation uses thread local storage that is relative to the `%fs` segment register):

```
1  ; Prologue:
2    mov    %fs:0x28,%rax
3    mov    %rax,-0x8(%rbp)
4    xor    %eax,%eax
5
6  ; Epilogue:
7    mov    -0x8(%rbp),%rcx
8    xor    %fs:0x28,%rcx
9    je     <safe_return>
10   callq <__stack_chk_fail@plt>
11 safe_return:
12   leaveq
13   ret
```

Listing 6.4: Prologue and epilogue for stack canaries.

6.4.3.2 Shadow stack

Shadow stacks are a strong form of stack integrity. The core idea is that sensitive data such as the return instruction pointer and any spilled stack pointers is moved to a second, protected stack. Any flaws in the program can therefore no longer corrupt the protected information. Code is instrumented to allocate

two stack frames for each function invocation: the regular stack frame and the shadow stack frame. The two stack frames can be of different size, i.e., the frame with the sensitive data may be much smaller than the regular stack frame. Figure 6.4 shows the shadow stack layout.

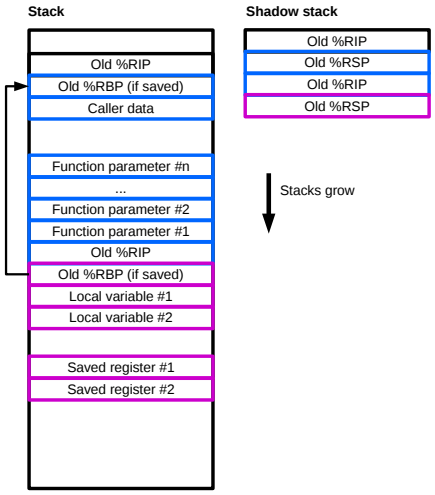


Figure 6.4: Shadow stack layout.

A key question is how the shadow stack is protected from arbitrary writes. Simply moving it to another memory location stops continuous buffer overflows (similarly to stack canaries) but cannot stop an adversary with arbitrary memory modification capabilities. Shadow stacks result in about 5% performance overhead due to the allocation of additional stack frames and checks when returning from a function.

6.4.3.3 Safe stack

Safe stacks are a form of stack integrity that reduce the performance penalty compared to shadow stacks. A shadow stack always keeps two allocated stack frames for each function invocation, resulting in overhead. Stack canaries are only added if unsafe buffers are in a stack frame. The goal of safe stacks is to achieve security guarantees of shadow stacks with low performance overhead by only executing checks for unsafe objects.

All variables that are accessed in a safe way are allocated on the safe stack. An optional unsafe stack frame contains all variables that *may* be accessed in an unsafe way. A compiler-based analysis infers if unsafe pointer arithmetic is used on objects or if references to local variables escape the current function. Any objects that are only accessed in safe ways (i.e., no odd pointer arithmetic and no reference to the object escapes the analysis scope) remain on the safe stack frame. Unsafe stack frames are only allocated when entering a function that contains unsafe objects. This reduces the amount of unsafe stack frame allocations, achieving low performance overhead while providing equal security to a safe shadow stack implementation. Figure 6.5 shows the safe stack layout.

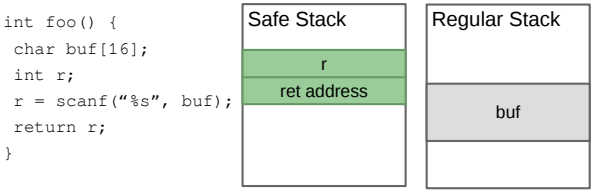


Figure 6.5: Safe stack layout.

6.4.4 Safe Exception Handling (SEH)

Programs often handle irregular control-flow, e.g., when error conditions are passed across several function call frames to where they are handled. While C allows `setjmp/longjmp` and `goto` for irregular control-flow, C++ provides a more elegant and structured variant to handle irregular control-flow: exceptions. C-style error handling is a crude tool to force control-flow to alternate locations. The high irregularity and immense flexibility (basically it is an unconditional jump across several contexts and stack frames completely under the control of the programmer) makes it impossible for the compiler to reason about its semantics and opens up opportunities for bugs. Exceptions are highly structured and allow the compiler to encode the necessary conditions when and how control-flow is transferred. Exception-safe code can safely recover from thrown conditions. Compared to C, the control-flow semantics are explicit in the programming language.

```
1 double div(double a, double b) {
2    if (b == 0)
3       throw "Division by zero!";
4    return (a/b);
5 }
6 ...
7 try {
8   result = div(foo, bar);
9 } catch (const char* msg) {
10   ...
11 }
```

Listing 6.5: Exception handling in C++

Exception handling requires support from the code generator (compiler) and the runtime system (libc or libc++). The implementation for exceptions is compiler-specific (libunwind for LLVM). When implementing exceptions, two different approaches exist: (a) inline exception information in stack frame or (b) generate exception tables that are used when an exception is thrown.

For *inline exception handling*, the compiler generates code that registers exceptions whenever a function is entered. Individual exception frames are linked (similar to a linked list) across stack frames. When an exception is thrown, the runtime system traces the chain of exception frames to find the corresponding handler. This approach is compact but results in overhead for each function call (as metadata about exceptions has to be allocated).

Exception tables trade-off per-function call costs to cost for each thrown exception. During code generation, the compiler emits per-function or per-object tables that link instruction pointers to program state with respect to exception handling. Throwing an exception is translated into a range query in the corresponding table, locating the correct handler for the exception. These tables are encoded very efficiently.

For both approaches, the encoding of the metadata may lead to security problems. Given a memory corruption vulnerability, an attacker can force throw an exception and may modify the way exceptions are handled by changing the exception data structures.

Microsoft Windows uses a combination of tables and inlined exception handling. Each stack frame records (i) unwinding information, (ii) the set of destructors that need to run, and (iii) the exception handlers if a specific exception is thrown. Unwinding information includes details on how the stack frame needs to be adjusted when an exception is thrown, e.g., what variables need to be stored from registers to memory or how the stack frame needs to be adjusted. An exception may close several scopes, resulting in objects going out of scope and therefore their destructors have to be run. When entering a function, a structured exception handling (SEH) record is generated, pointing to a table with address ranges for try-catch blocks and destructors. Handlers are kept in a linked list. To attack a Windows C++ program, an attacker may overwrite the first SEH record on the stack and point the handler to the first gadget. In response to this attack vector, Microsoft Visual Studio added two defenses: SafeSEH and SeHOP. SafeSEH

generates a compiler-backed list of allowed targets. If a record points to an unknown target it is rejected. SeHOP initializes the chain of registration records with a sentinel, i.e., the sentinel is the first element inserted on the linked list and therefore at the end of any exception list when an exception is thrown. If no sentinel is present, the handler is not executed. The two defenses guarantee that a set of benign targets is chained together ending with the sentinel but they do not guarantee that the right order of exceptions is executed nor the right number of exception handlers.

GCC encodes all exception information in external tables. When an exception is thrown, the tables are consulted to learn which destructors need to run and what handlers are registered for the current location of the instruction pointer. This results in less overhead in the non-exception case (as additional code is only executed *on-demand* but otherwise jumped over). The information tables can become large and heavyweight *compression* is used, namely an interpreter that allows on-the-fly construction of the necessary data. The efficient encoding has a downside: the interpreter of the encoding can be abused for Turing-complete computation [25].

6.4.5 Fortify Source

Format string vulnerabilities allow an attacker to read or write arbitrary memory locations. A format string vulnerability allows the adversary to control the first argument to a `printf` function. See Section 8.2.2 for more details on format string vulnerabilities.

To counter format string vulnerabilities, Microsoft simply deprecated the `%n` modifier. This stops the arbitrary write primitive but still allows the adversary to leak memory contents under format string vulnerabilities. For Linux, an extra check is added for format strings: (i) check for buffer overflows (i.e., only benign elements are accessed), (ii) check that the first argument is in a read-only area, and (iii) check if all arguments are used. Linux checks the following functions: `mem{cpy,pcpy,move,set}`, `str{n}cpy`, `stp{n}cpy`, `str{,n}cat`, `{,v}s{,n}printf`. The GCC/glibc fortify source patch distinguishes between four different cases: (i) known correct – do not check; (ii) not known if correct but checkable, i.e., compiler knows the length of the target – do check; (iii) known incorrect – compiler warning and do check; and (iv) not known if correct and not checkable – no check, overflows may remain undetected.

6.4.6 Control-Flow Integrity

Control-Flow Integrity (CFI) [1,4] is a defense mechanism that protects applications against control-flow hijack attacks. A successful CFI mechanism ensures that the control-flow of the application never leaves the predetermined, valid control-flow that is defined at the source code/application level. This means that an attacker cannot redirect control-flow to alternate or new locations.

CFI relies on a static, often compile-time analysis that infers the control-flow graph of the application. This analysis constructs

a set of valid targets for each indirect, forward edge, control-flow transfer. For example, a function pointer of type `void (*funcPtr)(int, int)` may only point to the functions in the program that match the prototype and are address taken. At runtime, CFI uses instrumentation to check if the observed value of the function pointer is in the set of statically determined valid targets. Figure 6.6 shows a CFI check and the target set (and target reduction).

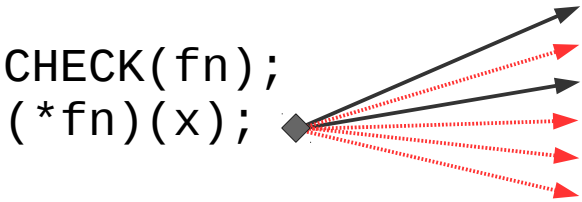


Figure 6.6: Control-Flow Integrity: at runtime, only valid targets (black arrows) are allowed, invalid targets (red arrows) result in termination.

Given indirect forward control-flow transfers (calls through function pointers in C/C++ or virtual dispatch in C++), what are valid targets of these control-flow transfers? A precise control-flow graph of the application lists all these valid targets but creating a precise control-flow graph is challenging due to aliasing, i.e., it is hard to infer the possible valid values of a code pointer through a static analysis. The best static CFI analysis would infer the precise set of targets for each location based on context and flow sensitive alias analysis, potentially with dynamic path tracking [4].

Due to the complexity of the analysis, existing CFI mechanisms

focus on alternate schemes to detect the sets of valid targets on a per-location basis. The simplest CFI analysis scheme simply uses the set of valid functions where any valid function can be called from any indirect control-flow transfer location. Another, more involved scheme counts the number of arguments and creates one set for each count, i.e., all functions without arguments, functions with 1 argument, functions with two arguments, and so on. The current state of the art for C CFI analysis leverages the function prototype and creates one set of targets per function prototype. For C, the scheme can be improved by measuring which functions are address taken. Only functions that are address taken and somewhere assigned to a function pointer can be used at runtime as pointer arithmetic on function pointers is undefined. For C++, this scheme is improved through class hierarchy analysis. The call site uses a certain type and, given a class hierarchy which must be available, only this type and subtypes in the inheritance chain are allowed for this call location.

```
0xf000b400

int bar1(int b, int c, int d);

int bar2(char *str);

void bar3(char *str);

void B::foo(char *str);

class Greeter :: Base {... };
void Base::bar5(char *str);

void Greeter::sayHi(char *str);

class Related :: Greeter {... };
void Related::sayHi(char *str);

Greeter *o = new Greeter();
o->sayHi(char *str);
```

Listing 6.6: Example of precision trade-offs for different CFI policies.

In the example above, let us look at the `sayHi` call in the last line. The valid function policy would allow all functions except the raw address `0xf000b400` which points somewhere into the code area (but not to a valid function). The arity policy would allow the set of `bar2`, `bar3`, `foo`, `Base::bar5`, `Greater::sayHi` , `Related::sayHi`. The function prototype

policy removes bar2 from the previous set, resulting in `bar3, foo, Base::bar5, Greater::sayHi , Related::sayHi`. Note that for C, this is the most precise prototype-based analysis possible. For the class hierarchy analysis, only the two functions `Greater::sayHi , Related::sayHi` are in the set, producing the smallest set of targets.

The different CFI analysis schemes provide a trade-off between security (precision) and compatibility. Given imprecise (unsafe) C code, the prototype-based check may fail for benign code. While this is an actual bug that should be fixed, some people argue that a mitigation should never prohibit benign code. Therefore, Microsoft uses the valid function policy for their Control-Flow Guard implementation of the CFI policy while Google uses the function prototype for C and the class hierarchy analysis for C++ code in the LLVM-CFI implementation.

CFI is an efficient mitigation to stop control-flow hijack attacks but is no panacea. CFI allows the underlying bug to fire and the memory corruption can be controlled by the attacker. The defense only detects the deviation after the fact, i.e., when a corrupted pointer is used in the program. Attackers are free to modify arbitrary data and can leverage complex programs to execute arbitrary computation without hijacking control flow. Alternatively, imprecision in the analysis allows attackers to choose arbitrary targets *in the set of valid targets* for each control-flow location.

6.4.7 Code Pointer Integrity

Code Pointer Integrity (CPI) [16] is a defense mechanism that protects applications against control-flow hijacking attacks. While memory safety and type safety would protect against all control-flow hijack attacks it results in a prohibitive overhead when enforced on top of low-level languages. Conceptually, memory safety protects code pointers against compromise. Memory safety and type safety protect the integrity of all pointers in a program. Unfortunately, memory safety and type safety result in prohibitive overhead.

The core idea of CPI is to restrict memory safety to sensitive pointers. Sensitive pointers are code pointers and pointers that, directly or indirectly, point to code pointers. Enforcing integrity (memory safety) for these pointers guarantees that a bug in the program cannot modify these pointers and thereby cannot hijack the control-flow. CPI is implemented as a compiler pass that moves sensitive pointers to a safe (sandboxed) memory area that is protected from adversarial access. Note that CPI does not enforce type safety for sensitive pointers.

6.4.8 Sandboxing and Software-based Fault Isolation

In various contexts both trusted and untrusted code must run in the same address space. The untrusted code must be sandboxed so that it cannot access any of the code or data of the trusted code while the trusted code may generally access code and data of the untrusted code. The untrusted code may only read/write its own data segment (and stack). Such

a compartmentalization primitive allows powerful use-cases, e.g., running a binary plugin in the address space of a browser without giving it access to the browser's sensitive data or mitigations (with potentially verified code) that keep sensitive values protected from the remaining large code base that may contain bugs.

On the 32-bit version of x86 segment registers allowed a separation of address spaces with segment registers enforcing a hard separation. Unfortunately, in the x86_64 extension, segment boundaries are no longer enforced.

Software-based Fault Isolation is a way to implement such a separation of the address space between trusted and untrusted parties. The memory access restriction can be implemented through masking each memory access with a constant: `and rax, 0x00ffffff; mov [rax], 0xc0fe0000`. The mask in the example restricts the write to the low 24 bit/16 MB of the address space.

Assuming that SFI is implemented by adding additional checks before a memory write then SFI could be bypassed by using an indirect jump to transfer control flow past the check but before the write. On CISC architectures, a jump may even transfer control into an instruction to execute an unintended instruction (e.g., on x86, `mov $0x80cd01b0, (%rax)` contains `mov $1, %al; int $0x80`). All indirect jumps therefore have to be aligned to valid instruction beginnings and for write instructions to before the check.

6.5 Summary

Several mitigations stop exploitation of unpatched or unknown memory and type safety vulnerabilities. Mitigations have low or negligible performance or runtime overhead due to the unquantified risk of bugs. Data Execution Prevention stops code injection attacks, but does not stop code reuse attacks. Address Space Layout Randomization is probabilistic, shuffles memory space and is prone to information leaks. Stack Canaries are probabilistic, do not protect against direct overwrites and are prone to information leaks. Safe Exception Handling protects exception handlers, reuse of handlers remains possible. Fortify source protects static buffers and format strings. These defenses fully mitigate code injection and probabilistically or partially mitigate code reuse and control-flow hijack attacks.

Novel defenses further increase the cost for an attacker to build a working exploit and reduce the chances of success. Shadow stacks enforce stack integrity and protect against return oriented programming. Control-Flow Integrity restricts forward-edge control-flow transfers to a small set. Sandboxing and Software-based Fault Isolation limit unsafe modules to a small area of code and/or data.

7 Case Studies

After discussing secure software development, security policies, software testing strategies, mitigations, and attack vectors, we will focus on several case studies that apply or refine the software security process. Software security is not static but must be adapted to a given situation to maximize protection.

We will evaluate two case studies: web security and mobile security. For web security, the attacker model includes both the server providing the web services and the browser or client running the web application. For mobile security, the central control over the market place allows the provider to thoroughly test apps before they are installed.

7.1 Web security

Web security is a broad topic that would deserve its own book due to the many aspects that need to be secured. In this Section, we will look at three broad aspects of web security: protecting the server, protecting the interaction between server and client, and protecting the web browser. Web servers (and browsers) are long running software applications that are exposed to adversaries as they serve external requests. Generally, software

daemons are long running services that serve external requests. Daemons often run in the background to provide functionalities to a system or, over the network, to multiple concurrent clients. A web server, mail server, or a DNS server are examples of daemons.

7.1.1 Protecting long running services

After initialization, daemons run for a long time, from days to weeks, to months without being restarted. Several mitigations such as ASLR or stack integrity rely on probabilistic guarantees where information is hidden from an adversary. Servers often run multiple threads that are restarted after handling a certain number of requests. Erring towards availability over security, crashing threads are restarted. An adversary can therefore leverage information leaks to recover secrets in a processes address space such as the memory layout to bypass ASLR or stack canary values to bypass stack canaries.

Daemons are complex as they serve requests in parallel through multiple threads. As such they are optimized for performance and leverage, e.g., distributed caches to reduce the per request cost, and they offer broad functionalities for different clients. This large complexity increases the risk for bugs. Complexity results in more code and code size correlates with the number of bugs in a given application.

As they are connected to the network, daemons are often exposed to the internet where they can be attacked from any remote location. The internet and the web are open platforms, allowing anyone to connect and request services. This

increases the risk that someone will launch a remote attack. Especially web servers provide functionality openly to users without requiring authentication or identification of the user.

A reasonable approach to reduce the attack surface of daemons is to break them into smaller components. These components then serve as fault compartments and fail independently. The overall idea is that when one component fails, others continue to function without opening security holes. Components communicate through well-defined APIs and, if one component is buggy, adversaries are restricted to the capabilities of the buggy component and must interact only with the privileges of that component. Without compartmentalization, the adversary would gain all privileges required by the service instead of a small subset.

A good example for compartmentalization are mail servers. Mail servers have a plethora of tasks: sending and receiving data from the network on a privileged port, parsing the mail protocol, managing a pool of received and unsent messages, providing access to stored messages for each user. The classic approach (implemented in sendmail) is to provide a single binary that executes all theses tasks. Due to the large amount of privileges required (full access to the file system, access to the network), the component runs as root with full administrator privileges. As the mail server accepts connections from the network, this results in a large attack surface and a prominent target that has been attacked frequently.

The modern approach (implemented in qmail) breaks the mailserver into a set of components that communicate with each other. Separate modules run as separate user IDs to

provide isolation. Each ID has only limited access to a subset of resources, enforcing least privilege. Only two components run as root and suid root respectively. The central component qmail-queue is privileged to run as qmailq user on behalf of arbitrary users. This small component allows anyone to register new mail in the mail queue. The qmail-send component received messages from the queue and either delivers them to qmail-rspawn to deliver them remotely or qmail-lspawn to deliver them locally. The qmail-lspawn component runs as root as it spawns a local delivery process with the correct target userid (of the receiving user). The qmail-local process runs on behalf of the target user and executes local mail filtering, spam filtering, and user-specific scripts. Note that this enables the mail server to allow customizable per-user filtering without exposing any attack surface. For incoming email, either an unprivileged network server listens for incoming messages and executes spam filtering and other tests or a local mail injection scripts passes messages to the qmail queue. Figure 7.1 gives an overview of the qmail system.

7.1.2 Browser security

Protecting a browser is similar to protecting a daemon. Browsers are long running processes (when have you last restarted your browser?). Through tabs, browsers run multiple contexts at the same time, often 10s or 100s of tabs are open concurrently and each tab must be isolated from each other.

A browser enables an interesting attacker model as the adversary can run JavaScript code on a victim's computer. The

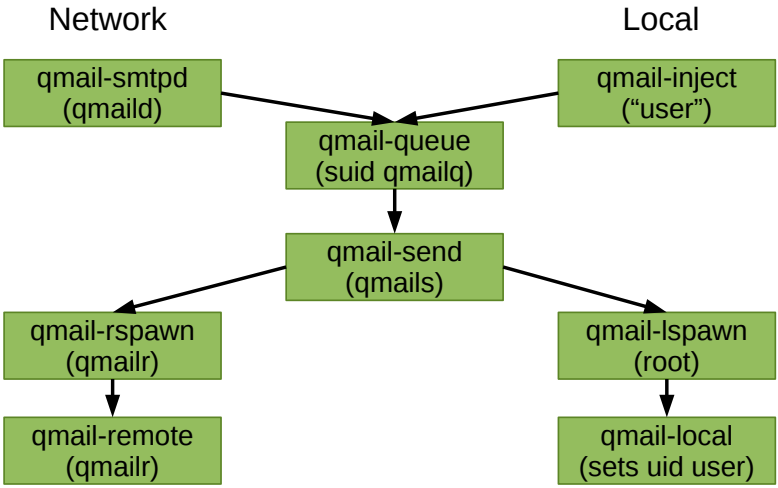


Figure 7.1: QMail mail server that is broken into minimally privileged components.

JavaScript compiler therefore needs to ensure that no information leaks from the process into the JavaScript sandbox.

Similar to mail servers discussed in the previous section, browsers can be implemented as single process with shared state/heap and multiple threads (Firefox) or broken into different processes (Chromium). For Chromium, each tab corresponds to an individual process. The complex and error prone task of parsing and rendering html is compartmentalized in an unprotected process with limited interaction capabilities to outside processes. This sandbox ensures that outside interactions are limited and restricted to a well-known API. Browsers are thoroughly tested to ensure that they follow strict security guidelines.

7.1.3 Command injection

The Unix philosophy is to leverage the combination of simple tools to achieve complex results. Many servers therefore leverage the `int system(char *cmd)` command to start new processes to execute simple Unix commands. Potentially tainted data from forms or user input is passed to these scripts or programs as parameters. The system command is an example where both code and data are mixed: both the command that is executed and the arguments are passed in a single string.

Dynamic web pages, for example, execute code on the server. This allows the web server to add rich content from other sources such as databases or files, providing dynamic content to the user. The dynamic web page executes as a script on the server side to collect information, build up the page, and

send it back to the user. A content management system may load content from the database, a set of files, and other remote parties, then intersect the information with the site template, add navigation modules, and send it back to the user.

```
1 <html><head><title>Display a file</title></head>
2 <body>
3 <? echo system("cat ".$_GET['file']); ?>
4 </body></html>
```

The PHP script above leverages the simple `cat` utility to return the contents of a user supplied file back to the user. Unfortunately, `system` executes a full shell, resulting in powerful command injection vulnerabilities. The arguments to system are the string `cat` concatenated with the user-supplied value in the parameter `file`, e.g., through `http://web.site/?file=user`. For example `;` allows chaining two commands such as `http://web.site/?file=user%3Bcat%20%2fetc%2fpasswd` to leak `/etc/passwd`. Simply blocking `;` is not enough, the user supplied data in file is untrusted and must be pruned either through validation (comparing against a set of allowed values) or escaping where the user-supplied values are clearly marked as string, e.g., resulting in `system("cat 'file; cat /etc/passwd'")` which would result in a file not found error. Note that you should not write your own escape functions, each web framework has their own escape functions that allow for different contexts. Even better, instead of leveraging `system`, simply open and read the file instead of launching a set of sub processes.

Command injection attacks are enabled through the fact that code and data share a communication channel. The `system` function expects a single string for both the command and the argument. A better system function would expect the command as first argument and all the arguments as the second argument. This is similar to code injection where, e.g., the stack can contain both code and data, allowing a buffer overflow to overwrite the instruction pointer to return to the stack.

7.1.4 SQL injection

SQL injection is similar to command injection: an SQL query contains both code and data. For example: `$sql = "SELECT * FROM users WHERE email='" . $_GET['email'] . "' AND pass='" . $_GET['pwd'] . ';"` creates an SQL query string with user input that can be sent to a database. Unfortunately, the user-supplied parameters are not escaped. An adversary may inject `'` characters to escape queries and inject commands. For example, an adversary may enter `asdf' OR 1=1 --` in the password field to bypass the password check.

To mitigate SQL injection, we apply the same idea: user-supplied arguments have to be validated or escaped. Alternatively, the control and data channel should be separated by using prepared SQL statements, similar to how `printf` defines a format string and with arguments that are then filled in: `sql("SELECT * FROM users WHERE email=\$1 AND pwd=\$2", email, pwd)`.

7.1.5 Cross Site Scripting (XSS)

Cross Site Scripting (or XSS) exploits trust user has in a web site. XSS enables an adversary to inject and execute JavaScript (or other content) in the context of another web page. For example, malicious JavaScript code may be injected into a banking web page to execute on behalf of a user that is logged into her bank account. This allows the adversary to extract username and password or to issue counterfeit transactions. There are three different kinds of XSS: persistent/stored, reflected, and client-side XSS.

Persistent XSS modifies data stored on the server to include JavaScript code. The adversary interacts with the web application, storing the code on the server side. When the user interacts with the web application, the server responds with a page that includes the attacker-supplied code. An example of persistent XSS is a simple chat application where the adversary includes `<script>alert('Hi there');</script>` in the chat message. This message is stored in a database on the server. When the message is sent to the user, the JavaScript code is executed on behalf of the user's browser in the user's session. Persistent XSS is enabled through a lack of input sanitization on the server side. Common locations of such errors are feedback forms, blog comments, or even product meta data (you do not have to see the response to execute it). In this scenario, the user only has to visit the compromised website.

Reflected XSS encodes the information as part of the request which is then reflected through the server back to the user. Instead of storing the JavaScript code on the server side, it

is encoded as part of the link that the user clicks on. A web interface may return the query as part of the results, e.g., "Your search for 'query' returned 23 results.". If the query is not sanitized, then JavaScript code will be executed on the user side. The code is encoded as part of the link and the user is tricked to click on the prepared link. The bug for this type of XSS is on the server side as well.

Client-side XSS targets lack of sanitization on the client side. Large web applications contain a lot of JavaScript code that also parses input data from, e.g., AJAX/JSON requests or even input that is passed through the server. This JavaScript code may contain bugs and missing sanitization that allows the adversary to execute injected JavaScript in the context of the web application as the user. Similarly to reflected XSS, the user must follow a compromised link. The server does not embed the JavaScript code into the page through server-side processing but the user-side JavaScript parses the parameters and misses the injected code. The bug is on the client side, in the server-provided JavaScript.

7.1.6 Cross Site Request Forgery (XSRF)

Cross Site Request Forgery (XSRF) exploits trust a web app has in user's browser. Given that a user is logged into a web page, certain links may trigger state changes on behalf of that user. For example, the URL "http://web.site/?post=Hello" may create a new public post on a bulletin board on behalf of the user with the content 'Hello'. An adversary that knows the URL pattern can construct URLs that trigger actions on

behalf of a user if they click on it. Instead of creating a public post, the action could be a money transfer from the user's bank account to the attacker.

7.2 Mobile security

Smart mobile devices have become ubiquitous: from smart phones to tablets and smart watches, they all run a form of mobile operating system that allows installation of apps from a central market. Android is one of two dominant ecosystems that cover the mobile platform. A hard challenge for Android is the large amount of different devices from different hardware vendors that is customized through different carriers. This results in a slow update process as all these different devices are hard to maintain and update.

The Android ecosystem enforces strict security policies that reduce the risk of running malicious apps. The basic Android system is configured to reduce the risk of software vulnerabilities by using a secure basic system configuration combined with strong mitigations. Individual apps are isolated from each other. Applications are installed from a central market. Each app in the market is vetted and tested through static and dynamic analysis to detect malicious code.

7.2.1 Android system security

Android is built on Linux and therefore follows a basic Unix system design. Instead of running multiple applications under

a single user id, each application is associated its own "user". Under the Unix design, each user has a home directory and associated files. Each application started by the user can access all the files of that user. Under Android, each app runs as its own user id and is therefore restricted to access only the files of that user. Interaction between apps is controlled through intents and a well-defined API.

The Android system leverages a hardened Linux kernel to protect against attacks from malicious apps. Apps are isolated from each other through the user id interface and the kernel is configured to reduce side channels through, e.g., the "/proc" interface. The filesystem follows a stringent set of permissions to reduce exposure and an SELinux policy enforces access control policies on processes. To protect against cold boot attacks or hardware attacks, Android leverages full filesystem encryption that is seeded from the user password. Services and daemons in user space leverage stack canaries, integer overflow mitigation, double free protection through the memory allocator, fortify source, DEP, ASLR, PIE, relro (mapping relocations as read-only after resolving them), and immediate binding (mapping the procedure linkage table as read-only after forcefully resolving all inter-module links). Each Android update includes security updates, patches, updates to the toolchain and tighter security defaults. Overall, Android follows a secure system default configuration and restricts interactions between apps and the system.

7.2.2 Android market

Linux distributions like Debian or Ubuntu have long provided a central market of curated applications. Developers provide their applications and package maintainers make sure that they fit well into the existing ecosystem of a distribution. Package maintainers are responsible to backport patches and ensure a smooth operation and integration between all the different applications that make up a distribution.

The Android mobile app market is similar and provides a central place where users can search for and install new apps. Developers sign apps (and their required permissions) and upload the apps to the market. Google can then vet and check apps before they are provided to individual users. Each application is tested for malicious code or behavior. Entrance to the market is regulated. Each developer must pay an entrance fee to upload apps. This entrance fee allows Google to offset the cost of the vetting process. If a malicious app is detected, all apps of a user can be blocked. This limits the amount of malicious code a user can upload and increases the risks for attackers that all their apps are blocked.

Whenever an app is updated and uploaded to the market, it is distributed to all devices that have it installed. This automatic update process minimizes the risk of exposure as the new version is pushed to the clients quickly.

7.2.3 Permission model

The app permission model restricts what devices an app has access to. The complex permission system allows a fine-grained configuration on a per-app basis on access to Camera, location information, Bluetooth services, telephony, SMS/MMS functionality, and network/data connections. Without privileges, an app is restricted to computation, graphics, and basic system access.

The user can select if they accept the permissions required by the app and if it matches the expected behavior. This model empowers the user to make security decisions. Whether the user is able to make informed decisions about security matters remains questionable. The user already searched for an app and is trying to install it. What are the chances that they will be negatively influenced through an over-privileged app? A better system to manage privileges remains to be found and is an active research area.

8 Appendix

The Appendix contains sections on reverse engineering, construction of shell code and ROP chains as well as details on some attack primitives. Currently, the appendix is only a stub but will be extended in the future.

8.1 Shellcode

Writing shellcode is an art that focuses on designing useful code that runs under tight constraints. Shellcode executes outside of a program context. Due to the missing context, shellcode can only use variables that are currently in registers, relative to the registers at the time of the exploit, at absolute addresses in the process, or leaked through a prior information leak.

The construction of shellcode often follows constraints of the vulnerability, e.g., only printable ASCII characters or no NULL bytes. To initialise state for the exploit, shellcode often uses tricks to recover state from the stack or relative addresses such as calling/popping into a register to recover the instruction pointer on x86 32-bit where the EIP register is not directly addressable.

8.2 ROP Chains

- Gadgets are a sequence of instructions ending in an indirect control-flow transfer (e.g., return, indirect call, indirect jump)
- Prepare data and environment so that, e.g., pop instructions load data into registers
- A gadget invocation frame consists of a sequence of 0 to n data values and a pointer to the next gadget. The gadget uses the data values and transfers control to the next gadget

Simple ROP tutorial

8.2.1 Going past ROP: Control-Flow Bending

- Data-only attack: Overwriting arguments to `exec()`
- Non-control data attack: Overwriting is admin flag
- Control-Flow Bending (CFB): Modify function pointer to valid alternate target
  - Attacker-controlled execution along valid CFG
  - Generalization of non-control-data attacks
  - Each individual control-flow transfer is valid
  - Execution trace may not match non-exploit case

Control-Flow Bending research paper

8.2.2 Format String Vulnerabilities

Functions that handle format strings (e.g., the `printf` family) allow a flexible configuration of the printed data through the first argument (the format string). If an attacker can control the format string then they can often achieve full arbitrary read and write primitives.

9 Acknowledgements

I would like to thank several people for feedback on the different drafts of the book, for suggesting new topics to include, for pointing out typos, and other valuable help in improving this text!

- Anders Aspnäs for feedback on practical implications of software engineering techniques;
- Nathan Burow for spelling and writing issues;
- Joachim Desroches for spelling and writing issues;
- David Dieulivol for spelling and writing issues;
- Frédéric Gerber for spelling and writing issues;
- Debbie Perouli for several writing suggestions;
- Josua Stuck for spelling and writing issues.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the acm conference on computer and communications security.* DOI:https://doi.org/10.1145/1102120.1102165

[2] Matt Bishop. 2002. Robust programming.

[3] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassman, and Anna Shubina. 2011. Exploit programming. From buffer overflows to "weird machines" and theory of computation. *Usenix ;login:* (2011).

[4] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. In *ACM Computing Surveys.* DOI:https://doi.org/10.1145/3054924

[5] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 acm on asia conference on computer and communications security.*

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage

tests for complex systems programs. In *Proceedings of the usenix conference on operating systems design and implementation.*

[7] Google. 2010. Google unit testing framework. Retrieved from https://github.com/google/googletest

[8] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Herbert Bos, Cristiano Giuffrida, and Erik van der Kouwe. 2016. TypeSanitizer: Practical Type Confusion Detection. In *Proceedings of the acm conference on computer and communications security.* DOI:https://doi.org/10.1145/2976749.2978405

[9] Michael Hicks. 2014. What is memory safety? Retrieved July 21, 2014 from http://www.pl-enthusiast.net/2014/07/21/memory-safety/

[10] Grace Hopper. 1947. Software bug etymology.

[11] Yuseok Jeon, Priyam Biswas, Scott A. Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the acm conference on computer and communications security.* DOI:https://doi.org/10.1145/2976749.2978405

[12] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of c. In *Usenix atc '02.*

[13] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 acm sigsac conference on computer and communications security.*

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd acm symposium on operating systems principles*, 207–220. DOI:ht tps://doi.org/10.1145/1629575.1629596

[15] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the thirteenth eurosys conference*.

[16] Volodymyr Kuzentsov, Mathias Payer, Laszlo Szekeres, George Candea, Dawn Song, and R. Sekar. 214AD. Code Pointer Integrity. In *Operating systems design and implementation*.

[17] Butler W. Lampson. 1974. Protection. *ACM Operating Systems Review* (1974).

[18] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *Usenix security symposium*.

[19] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52, 7 (2009), 107–115. Retrieved from http://gallium.inria.fr/~xleroy/publi/compcer t-CACM.pdf

[20] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *SNAPL*.

[21] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for c. In *ACM conference on programming languages design and implementation*.

[22] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ACM international symposium on memory management*.

[23] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* (2005).

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th acm sigplan conference on programming language design and implementation*. DOI:https://doi.org/10.1145/1250734.1250746

[25] James Oakland and Sergey Bratus. 2011. Exploiting the hard-working dwarf. In *Usenix workshop on offensive technologies*.

[26] Mathias Payer. 2012. Too much PIE is bad for performance. In *Technical report http://nebelwelt.net/publications/files/12TRpie.pdf*.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-fuzz: Fuzzing by program transformation. In *IEEE international symposium on security and privacy, 2018*.

[28] Google Chromium Project. 2013. Undefined behavior sanitizer.

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the network and distributed system security symposium (ndss)*.

[30] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX atc 2012*.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*.

[32] LLVM team. 2018. Libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[33] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *ACM symposium on operating systems principles*. DOI:https://doi.org/10.1145/168619.168635

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE symposium on security and privacy*. DOI:https://doi.org/10.1109/SP.2009.25

[35] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution

engine tailored for hybrid fuzzing. In *27th usenix security symposium (usenix security 18)*.

[36] Michal Zalewski. 2014. Technical whitepaper for afl-fuzz. Retrieved from http://lcamtuf.coredump.cx/afl/technical_details.txt