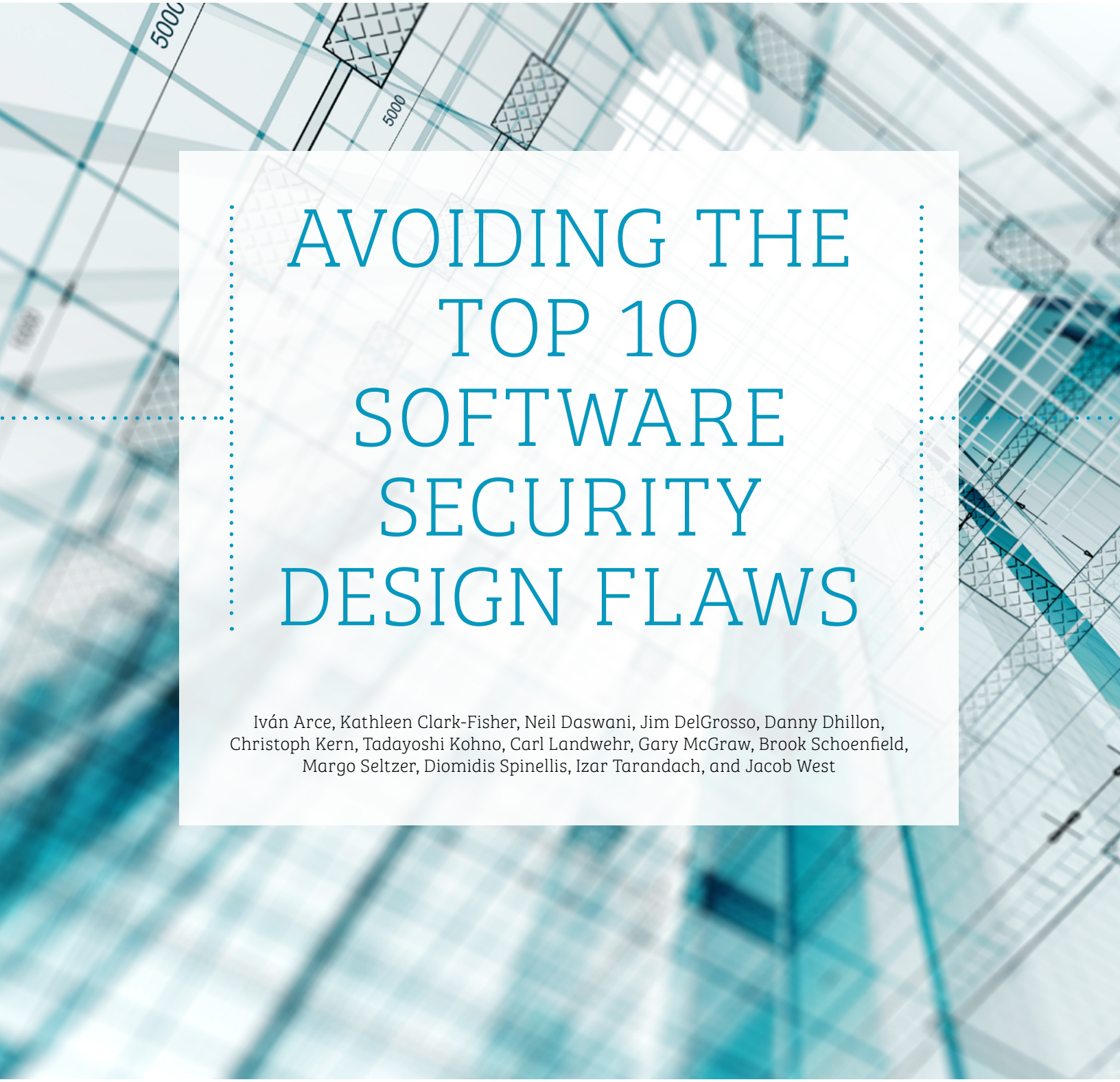
IEEE CENTER FOR SECURE DESIGN



AVOIDING THE TOP 10 SOFTWARE SECURITY DESIGN FLAWS

Iván Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West

IEEE

IEEE computer society

IEEE CYBER SECURITY

read on

CONTENTS

IEEE Computer Society Center for Secure Design Participants

Iván Arce, Sadosky Foundation

Neil Daswani, Twitter

Jim DelGrosso, Cigital

Danny Dhillon, RSA

Christoph Kern, Google

Tadayoshi Kohno, University of Washington

Carl Landwehr, George Washington University

Gary McGraw, Cigital

Brook Schoenfield, McAfee, Part of Intel Security Group

Margo Seltzer, Harvard University

Diomidis Spinellis, Athens University of Economics and Business

Izar Tarandach, EMC

Jacob West, HP

Staff

Kathleen Clark-Fisher, Manager, New Initiative Development

Jennie Zhu-Mai, Designer

Public Access Encouraged

About the IEEE Computer Society

The IEEE Computer Society is the world's leading computing membership organization and the trusted information and career-development source for a global workforce of technology leaders. The Computer Society provides a wide range of forums for top minds to come together, including technical conferences, publications, and a comprehensive digital library, unique training webinars, professional training, and the TechLeader Training Partner Program to help organizations increase their staff's technical knowledge and expertise. To find out more about the community for technology leaders, visit http://www.computer.org.

Published by the IEEE Computer Society.

INTRODUCTION

Most software that has been built and released typically comes with a set of defects—implementation bugs and design flaws. To date, there has been a larger focus on finding implementation bugs rather than on identifying flaws.

In 2014, the IEEE Computer Society, the leading association for computing professionals, launched a cybersecurity initiative with the aim of expanding and escalating its ongoing involvement in the field of cybersecurity. The first step for the initiative was to launch the IEEE Computer Society Center for Secure Design. The Center intends to shift some of the focus in security from finding bugs to identifying common design flaws in the hope that software architects can learn from others' mistakes. To achieve this goal, the Center brought people together from different organizations at a workshop in early 2014.

At the workshop, participants discussed the types of flaws they either identified in their own internal design reviews, or that were available from external data. They arrived at a list they felt were the top security design flaws. Many of the flaws that made the list have been well known for decades, but continue to persist. In this document is the result of that discussion—and how to avoid the top 10 security flaws.

IEEE
CENTER FOR
SECURE DESIGN

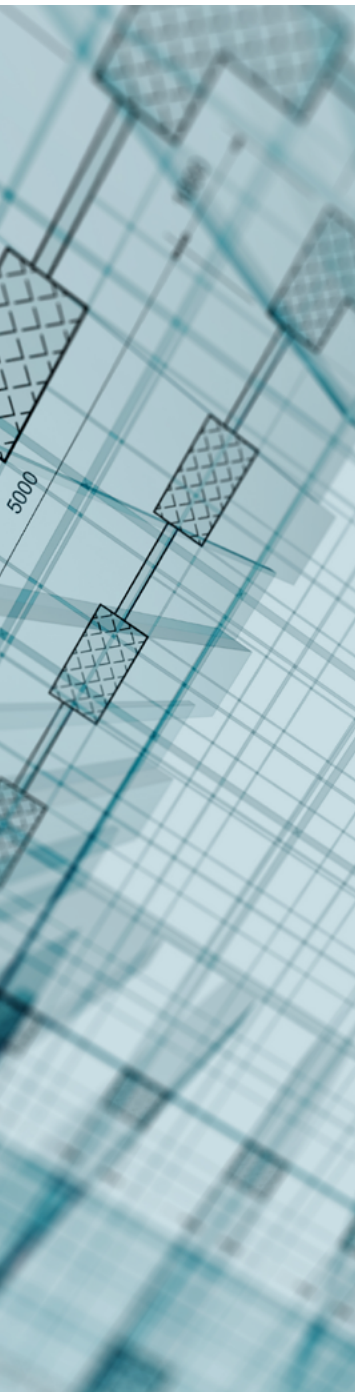
MISSION STATEMENT

The IEEE Computer Society's Center for Secure Design (CSD) will gather software security expertise from industry, academia, and government. The CSD provides guidance on:

- Recognizing software system designs that are likely vulnerable to compromise.
- Designing and building software systems with strong, identifiable security properties.

The CSD is part of the IEEE Computer Society's larger cybersecurity initiative, launched in 2014.

IEEE IEEE computer society IEEE CYBER SECURITY



PREAMBLE

The goal of a secure design is to enable a system that supports and enforces the necessary authentication, authorization, confidentiality, data integrity, accountability, availability, and non-repudiation requirements, even when the system is under attack.

While a system may always have implementation defects or "bugs," we have found that the security of many systems is breached due to design flaws or "flaws." We believe that if organizations design secure systems, which avoid such flaws, they can significantly reduce the number and impact of security breaches.

While bugs and flaws are both different types of defects, we believe there has been quite a bit more focus on common bug types than there has been on secure design and the avoidance of flaws. Before we discuss our contribution in this document, we briefly discuss the differences between bugs and flaws.

Both bugs and flaws are types of defects. A defect may lie dormant in software for years only to surface in a fielded system with major consequences. A bug is an implementation-level software problem. Bugs may exist in code but never be executed. A flaw, by contrast, is a problem at a deeper level. Flaws are often much more subtle than simply an off-by-one error in an array reference or use of an incorrect system call. A flaw might be instantiated in software code, but it is the result of a mistake or oversight at the design level. For example, a number of classic flaws exist in error-handling and recovery systems that fail in an insecure or inefficient fashion.

In this document, a group of software security professionals have contributed both real-world data and expertise to identify some of the most significant design flaws that have led to security breaches over the past several years. The list of issues presented here is focused entirely on the most widely and frequently occurring design flaws as compiled from data provided by the member organizations of the IEEE Computer Society Center for Secure Design (CSD).


5000
5000

EARN OR GIVE, BUT NEVER ASSUME, TRUST

Software systems comprising more than just a single monolithic component rely on the composition and cooperation of two or more software tiers or components to successfully accomplish their purpose. These designs often depend on the correct functioning of the existing parts. They will be inherently insecure if any of those parts are run in a potentially hostile environment, such as a user's desktop computer, an unmanaged device, or a runtime or sandbox that can be tampered with by an attacker.

Offloading security functions from server to client exposes those functions to a much less trustworthy environment, which is one of the most common causes of security failures predicated on misplaced trust.

Designs that place authorization, access control, enforcement of security policy, or embedded sensitive data in client software thinking that it won't be discovered, modified, or exposed by clever users or malicious attackers are inherently weak. Such designs will often lead to compromises.

Classic examples of software where trust is misplaced include a web browser or a thick-client application, but there are many more examples of client software. They include applications running on a mobile device, or embedded software that might be found in modern automobiles, pacemakers, gaming systems, or home appliances. Even calls into your APIs from business partners could be considered client software in some sense.

When untrusted clients send data to your system or perform a computation on its behalf, the data sent must be assumed to be compromised until proven otherwise. In some cases you may be able to guarantee that the client is, indeed, who it attests it is, or that the business logic it contains has not been altered or circumvented, or that external factors have not influenced the integrity of the computations it performed. But these situations are not the rule, and these underlying assumptions can change when new vulnerabilities are discovered. It is safer in the long run to design a software system under the assumption that components running on any platform whose integrity can't be attested are inherently not trustable, and are therefore unsuitable for performing security sensitive tasks.

If, nonetheless, security operations must be offloaded to components running on an untrusted platform, the design should impose extreme caution on how the computation and its output are treated.

Common weaknesses related to client trust reside in various parts of the system, but tend to share a sensibility. A designer might

Make sure all data received from an untrusted client are properly validated before processing.

(incorrectly) assume that server APIs will always be called in the same order every time. He or she might believe that the user interface is always able to restrict what the user is able to send to the server. He could try to build the business logic solely on the client side, or attempt to actually store a secret in the client. And, of course, a designer can run into danger by thinking that *any* intellectual property (IP) sent to the client can be protected through technical means.

Though security-aware development strategies cannot eliminate all these problems (or even resolve conflicts in goals for the software being developed), there are useful ways to minimize the potential risks. For example, some organizations will claim a real business need to store intellectual property or other sensitive material on the client. The first consideration is to confirm that sensitive material really does need to be stored on the client. When it truly is necessary to do so, various binary protection mechanisms can delay the leaking of sensitive material. Possible techniques to consider include obfuscation or anti-debugging (although the strength of these protections vary widely, so designers should understand the level of protection actually achieved with each tool or technique). Subject matter experts should be consulted if the system requires a client component with a level of protection that cannot be trivially compromised.

If IP or sensitive material must be stored or sent to the client, the system should be designed to be able to cope with potential compromise. For instance, the same shared secret or other cryptographic material shouldn't be used on all the clients. Make the validity of what is offloaded to the client limited in time, set expiration dates for data stored in the client, watermark IP, and double-check client computations that are security sensitive. On a related note, design your system to work in a limited fashion even when one or many clients have been completely compromised.

Finally, make sure all data received from an untrusted client are properly validated before processing. Follow the guidance described in the "Define an Approach that Ensures All Data Are Explicitly Validated" section.

When designing your systems, be sure to consider the context where code will be executed, where data will go, and where data entering your system comes from. Failing to consider these things will expose you to vulnerabilities associated with trusting components that have not earned that trust.

USE AN AUTHENTICATION MECHANISM THAT CANNOT BE BYPASSED OR TAMPERED WITH

Authentication is the act of validating an entity's identity. One goal of a secure design is to prevent an entity (user, attacker, or in general a "principal") from gaining access to a system or service without first authenticating. Once a user has been authenticated, a securely designed system should also prevent that user from changing identity without re-authentication.

Authentication techniques require one or more factors such as: something you know (e.g., a password), something you are (e.g., biometrics such as fingerprints), or something you have (e.g., a smartphone). Multi-factor (sometimes referred to as N-factor) authentication refers to the technique of requiring multiple distinct factors to prove your identity. Authentication via a cookie stored on a browser client may be sufficient for some resources; stronger forms of authentication (e.g., a two-factor method) should be used for more sensitive functions, such as resetting a password.

In general, a system should consider the strength of the authentication a user has provided before taking action. Note also that authentication encompasses more than just human-computer interaction; often, in large distributed systems, machines (and/or programs running on those machines) authenticate themselves to other machines.

The ability to bypass an authentication mechanism can result in an unauthorized entity having access to a system or service that it shouldn't. For example, a system that has an authentication mechanism, but allows a user to access the service by navigating directly to an "obscure" URL (such as a URL that is not directly linked to in a user interface, or that is simply otherwise "unknown" because a developer has not widely published it) within the service without also requiring an authentication credential, is vulnerable to authentication bypass.

The use of authentication techniques that don't fall into the category of something you know, something you are, or something you have may also allow users to access a system or service they shouldn't. System designers should beware of authentication techniques that depend on assumptions about sole possession of resources that may actually be shared. For example, authentication mechanisms that identify a user by their IP address wouldn't be useful if the addresses were shared among different users at different times; for instance, via an address-sharing/configuration protocol such as DHCP.

Even when IP addresses are tied to particular devices, authentication based on device addresses is not a substitute for user authentication, as IP addresses can be spoofed and are not necessarily associated with specific users for a long time. As another concrete illustration, authentication mechanisms that rely on a computer's MAC address, which can easily be changed or spoofed, can result in unauthorized access if the device assumed to be identified with that individual is lost or stolen.

Typically, the act of authentication results in the creation of a token, capability (as often referred to in operating systems literature), or ticket representing a principal that is used throughout the system or service. If such tokens (or credentials) are deterministically derived from easy-to-obtain information, such as a user name, then it becomes possible to forge identities, allowing users to impersonate other users.

Credentials must not be easy to forge. Upon successful authentication, the user may be provided with an authentication credential, token, or ticket, which can be provided back to the system so that the user does not need to be re-authenticated for every request or transaction made via the system. At the same time, if it is possible for an attacker to forge the authentication credential, token, or ticket, the attacker can bypass the authentication mechanism. System designers can reuse time-tested authentication mechanisms such as Kerberos instead of building a new one. Alternatively, system designers are encouraged to use cryptography correctly (see the corresponding "Using Cryptography Correctly" section later in this document) in constructing authentication credentials, tokens, and tickets.

If an authentication system does not limit the lifetime of an authentication interaction, then it may inadvertently grant access to a user to whom it should not. For example, imagine a user who logs into a public terminal and then walks away without logging out (which should terminate the session). A second user using the public terminal might now be able to use the system or service as the first user. A properly designed authentication system may automatically log the user out after a period of inactivity.

Authentication system designs should automatically provide a mechanism requiring re-authentication after a period of inactivity or prior to critical operations. As an example, upon receiving a transaction request to conduct certain sensitive actions such as changing a password, or transferring funds to another financial institution, a system could ask the user to re-enter their existing password again to confirm their transaction request, even though the user may already be authenticated.

The design of a system's re-authentication scheme, and when and how often to ask a user to re-enter their password, needs to be mindful of not only security, but also usability and convenience. Asking users to frequently re-enter their password can be damaging to security, as it trains people's muscle memory to enter their password every time they see a prompt and sets them up as easy phishing targets.

By far the most common authentication mechanism remains the password. Using passwords requires that the system or service have a mechanism to associate a given password with a particular user. If this information is not properly stored, it may be possible for agents other than the user to obtain access to them. Storing such information securely is non-trivial, and the reader is referred to the use of an applied cryptography expert as noted in the "Using Cryptography Correctly" section for guidance. Just as it is advisable to reuse tried and tested cryptographic algorithms, it is also advisable to re-use already built and tested password management systems instead of building new ones.

It's preferable to have a single method, component, or system responsible for authenticating users. Such a single mechanism can serve as a logical "choke point" that cannot be bypassed. Much as in code reuse, once a single mechanism has been determined to be correct, it makes sense to leverage it for all authentication.

To summarize, authentication mechanisms are critical to secure designs. They can be susceptible to various forms of tampering and may potentially be bypassed if not designed correctly. We recommend that a single authentication mechanism leverage one or more factors as per an application's requirements, that it serve as a "choke point" to avoid potential bypass, and that authentication credentials have limited lifetimes, be unforgeable, and be stored so that if the stored form is stolen, they cannot easily be used by the thief to pose as legitimate users.

AUTHORIZE AFTER YOU AUTHENTICATE

While it is extremely important to assess a user's identity prior to allowing them to use some systems or conduct certain actions, knowing the user's identity may not be sufficient before deciding to allow or disallow the user to perform certain actions. For instance, once an automatic teller machine (ATM) authenticates a user via something they have (a debit card), and something they know (a PIN), that does not necessarily mean that user is allowed to withdraw an arbitrary amount of cash from their account. Most users may be authorized to withdraw up to a certain limit per day, or to conduct certain actions (view balance) but not others (transfer funds outside the bank) from the ATM.

Authorization should be conducted as an explicit check, and as necessary even after an initial authentication has been completed. Authorization depends not only on the privileges associated with an authenticated user, but also on the context of the request. The time of the request and the location of the requesting user may both need to be taken into account.

Sometimes a user's authorization for a system or service needs to be revoked, for example, when an employee leaves a company. If the authorization mechanism fails to allow for such revocation, the system is vulnerable to abuse by authenticated users exercising out-of-date authorizations.

For particularly sensitive operations, authorization may need to invoke authentication. Although authorization begins only after authentication has occurred, this requirement is not circular. Authentication is not binary—users may be required to present minimal (such as a password) or more substantial (e.g. biometric or token-based) evidence of their identity, and authentication in most systems is not continuous—a user may authenticate, but walk away from the device or hand it to someone else. Hence authorization of a specially sensitive operation (for example, transferring a sum of money larger than a designated threshhold) may require a re-authentication or a higher level of authentication. Some policies require two people to authorize critical transactions ("two-person rule"). In such cases, it is important to assure that the two individuals are indeed distinct; authentication by password is insufficient for this purpose.

Finally, just as a common infrastructure (e.g., system library or back end) should be responsible for authenticating users, so too should common infrastructure be re-used for conducting authorization checks.

STRICTLY SEPARATE DATA AND CONTROL INSTRUCTIONS, AND NEVER PROCESS CONTROL INSTRUCTIONS RECEIVED FROM UNTRUSTED SOURCES

Co-mingling data and control instructions in a single entity, especially a string, can lead to injection vulnerabilities. Lack of strict separation between data and code often leads to untrusted data controlling the execution flow of a software system. This is a general problem that manifests itself at several abstraction layers, from low-level machine instructions and hardware support to high-level virtual machine interpreters and application programming interfaces (APIs) that consume domain-specific language expressions.

"At lower layers, lack of strict segregation between data and control instructions can manifest itself in memory-corruption vulnerabilities, which in turn may permit attacker-controlled modifications of control flow or direct execution of attacker-controlled data as machine or bytecode instructions."

At higher levels, co-mingling of control and data often occurs in the context of runtime interpretation of both domain-specific and general-purpose programming languages. In many languages, control instructions and data are often segregated using in-band syntactic constructs, such as quoting and escaping. If software assembles a string in a parseable language by combining untrusted data with trusted control instructions, injection vulnerabilities arise if the untrusted data are insufficiently validated or escaped. In that situation, an attacker may be able to supply data crafted such that when the resulting expression is processed, parts of the data are parsed and interpreted as control (rather than uninterpreted data, as intended). Experience has shown that use of injection-prone APIs incurs significant risk that injection vulnerabilities will indeed be introduced. Examples of such vulnerabilities include SQL query injection, cross-site JavaScript injection, and shell command injection.

At lower levels, software platforms can utilize hardware capabilities to enforce separation of code and data. For example, memory access permissions can be used to mark memory that contains only data as non-executable and to mark memory where code is stored as executable, but immutable, at runtime. Modern operating systems take advantage of such hardware features to implement security mechanisms that harden the entire software stack against multiple forms of attack. Software designs that ignore the principle of strict separation between data and code, or that blur the line that distinguishes one from the other, are inherently

less secure because they undermine or directly invalidate low-level security mechanisms.

When designing languages, compilers, virtual machines, parsers and related pieces of infrastructure, consider control-flow integrity and segregation of control and potentially untrusted data as important design goals.

When designing APIs (both general-purpose or public interfaces as well as those that are domain- or application-specific), avoid exposing methods or endpoints that consume strings in languages that embed both control and data. Prefer instead to expose, for example, methods or endpoints that consume structured types that impose strict segregation between data and control information.

When designing applications that rely on existing APIs, avoid APIs that mingle data and control information in their parameters, especially when those parameters are strings. If there is no choice in underlying APIs (for example, if the use of a relational database requires interfacing through a SQL query API), it is often desirable to encapsulate the injection-prone interface and expose its functionality to application code through a higher-level API that enforces strict segregation between control statements and potentially untrusted data.

A design that relies on the ability to transform data into code should take special care to validate the data as fully as possible and to strictly constrain the set of computations that can be performed using data as an input language. Specific areas of concern include the `eval` function, query languages, and exposed reflection.

`Eval.` Many interpreted languages (such as Python, Ruby, and JavaScript) have an `eval` function that consumes a string consisting of syntax in that language and invokes the language's interpreter on the string. Use of a language's `eval` facility can permit the implementation of very powerful features with little code, and is therefore tempting. It is also very dangerous. If attackers can influence even part of a string that is evaluated and that substring is not appropriately validated or encoded, they can often execute arbitrary code as a result.

Query languages. Ensuring that appropriate validation or escaping is consistently applied in all code that interfaces with the query API is a difficult and error-prone process; implementing that functionality repeatedly increases the risk of injection vulnerabilities. Use or develop an API that mediates between application code and raw query-language based interfaces (such as SQL, LDAP) and exposes a safer API. Avoid code that constructs queries based on ad-hoc string concatenation of fixed query stanzas with potentially untrusted data.

Exposed reflection. Many programming languages provide facilities that allow programs to reflectively inspect and manipulate objects, as well as to invoke methods on objects. Use of reflection can be very powerful, and often permits the implementation of complex features using minimal code. For example, implementations of object serializers and deserializers used to marshal and unmarshal in-memory objects into and from a serialized form for persistence or network transfer can often be implemented very effectively using reflection.

However, as with `eval`, use of reflection can be a risky design choice. Unless inputs processed with reflection are very carefully controlled, bugs can arise that may permit the attacker to execute arbitrary code in the receiving process. It is often preferable to consider alternative, safer designs. For example, consider a design based on code-generation: a code-generated, reflection-free object serializer/deserializer is restricted to behaviors allowed by the explicitly generated code. This code is in turn generated at build/compile-time, where the code-generation process cannot be influenced by malicious inputs.

DEFINE AN APPROACH THAT ENSURES ALL DATA ARE EXPLICITLY VALIDATED

Software systems and components commonly make assumptions about data they operate on. It is important to explicitly ensure that such assumptions hold: vulnerabilities frequently arise from implicit assumptions about data, which can be exploited if an attacker can subvert and invalidate these assumptions.

As such, it is important to design software systems to ensure that comprehensive data validation actually takes place and that all assumptions about data have been validated when they are used.

It is furthermore desirable to design software to make it feasible for a security reviewer to effectively and efficiently reason about and verify the correctness and comprehensiveness of data validation. Designing for verifiability should take into account that code typically evolves over time, resulting in the risk that gaps in data validation are introduced in later stages of the software life-cycle.

Design or use centralized validation mechanisms to ensure that all data entering a system (from the outside) or major component (from another component of the same system) are appropriately validated. For example:

- It is desirable for web applications to utilize a mechanism (such as a request filter or interceptor facility provided by the underlying web application framework) to centrally intercept all incoming requests, and to apply basic input validation to all request parameters.
- Implementations of communication protocols might centrally validate all fields of all received protocol messages before any actual processing takes place.
- Systems consuming complex data formats (such as XML documents, image file formats, or word processing file formats) might perform parsing, syntactic validation, and semantic validation of input files in a dedicated validation module whose output is a validated internal object representation of the input document. Parsers and validators must themselves be designed to robustly cope with potentially malicious or malformed inputs.

Transform data into a canonical form, before performing actual syntactic or semantic validation. This ensures that validation cannot be bypassed by supplying inputs that are encoded in a transport encoding, or in a possibly invalid non-canonical form.

Use common libraries of validation primitives, such as predicates that recognize well-formed email addresses, URLs, and so forth. This ensures that all validation of different instances of the same type of data applies consistent validation semantics. Consistent use of common validation predicates can also increase the fidelity of static analysis. Validation should be based on a whitelisting approach, rather than blacklisting.

Input validation requirements are often state-dependent. For instance, in a stateful protocol, the set of valid values of a particular protocol message field (and hence the corresponding validation requirements) may depend on the protocol's state. In such scenarios, it can be beneficial to design the protocol implementation's input validation component to be itself state-aware.

Explicitly re-validate assumptions "nearby" code that relies on them. For example, the entry points of a web application's business-logic layer should explicitly re-state, and check as preconditions, all assumptions that it relies on. Liberal use of precondition checks in the entry points of software modules and components is highly recommended. Such precondition checks should never fail during execution of the deployed application, assuming the higher layers of the application have correctly validated external inputs. And as such, it is unnecessary for the business-logic layer to produce friendly error messages should such a precondition fail. Nevertheless, re-validation of data supplied to the business-logic layer provides two benefits:

- It protects against vulnerabilities that arise from insufficient input validation in a higher layer (since the developer of the higher layer may not have a full understanding of all the requirements and assumptions of the lower layer), or from additional data-flows that were not considered during the initial security design (e.g., a data-load job that calls the business layer with data read from a file format used to exchange information between affiliated organizations, and which does not perform the same level of data validation as the web front end, based on the possibly invalid assumption that such files are "trusted").
- It permits local reasoning about the correctness of a component; since assumptions are explicitly checked and stated, a human reviewer or static analysis tool can truly assume the assumptions actually hold, without having to consider all (possibly very complex) data flows into the component.

Use implementation-language-level types to capture assumptions about data validity. For example, an application that receives as an input a date and time in string representation should validate that this input indeed consists of a well-formed string representation of a date and time (for example, in ISO 8601 format). It is desirable to implement validation by parsing the input into a typed representation (such as a "date and time" type provided in many programming language's standard libraries), and to use that typed representation (and not the original input string) throughout the program. Downstream components are then relieved from having to consider the possibility that a provided value (such as a date) is syntactically invalid, and can focus on only checking additional preconditions that are not supported by the type's contract (e.g., that a date is not in the future).

Various problems arise from failure to address this security design principle.

- Injection vulnerabilities can arise if untrusted data are used without validation in certain contexts, such as APIs and platform features that process and interpret strings with certain semantics. For example:
  - Using an externally controlled string as a component of a file path can lead to path traversal vulnerabilities, unless the application validates that the input represents a single path component (and, in particular, does not contain path separators).

Typically, security vulnerabilities in this category are highly domain- and application-specific.

  - If an externally controlled string is used in a context in a HTML document where it will be interpreted as a URL, a Cross-Site Scripting (XSS) vulnerability can arise unless it has been validated that the string represents a well-formed URL with a benign scheme (such as http:/ https:, and, in particular, not javascript:, vbscript:, data:, or others).
- It is generally preferable to perform data validation relevant to the prevention of injection vulnerabilities in the implementation of the API that is subject to injection vulnerabilities, or in a wrapper API in case the underlying API cannot be modified. See also "Strictly Separate Data and Control Instructions, and Never Process Control Instructions Received from Untrusted Sources" section.
- Attempting to validate data that are not in canonical form can allow validation to be bypassed. For example, it is difficult to validate that an input string represents a single path component (free of path separator characters) unless the input has been fully decoded (with respect to transport encodings) and has been validated to be in a canonical character encoding—otherwise, it might be possible for an attacker to sneak a path separator past the input validation by representing it in an encoded form (for example, %-encoding commonly used in web applications), or in the form of a non-canonical character encoding (for example, a non-canonical UTF-8 encoding).
- In applications implemented in non-memory safe languages such as C, failing to carefully validate external inputs can result in memory corruption vulnerabilities such as buffer overflows, unbounded memory reads, null-terminated string issues, and so on.
- Accepting inputs from untrusted sources without enforcement of an upper bound on data size can result in resource exhaustion.
- In general, aside from memory corruption and resource exhaustion issues, data that are not validated cause security issues primarily when they are used in a way that influences control flow. Data that are simply being copied around (e.g., received from an external input, then stored in a database, and later displayed in UI) are generally harmless. Problems arise if the application inspects the data and makes control flow decisions based on the data's value. This most immediately applies to data that are used in contexts where they are interpreted as instructions or control, leading to injection vulnerabilities as discussed earlier.

  More generally however, control-flow dependencies on untrusted, non-validated data can lead to state corruption vulnerabilities, or execution of state transitions that the programmer did not intend or consider. Typically, security vulnerabilities in this category are highly domain- and application-specific, and hence are difficult to reason about and detect by general-purpose tools. Careful, state-dependent validation of inputs can go a long way toward mitigating this risk.

USE CRYPTOGRAPHY CORRECTLY

Cryptography is one of the most important tools for building secure systems. Through the proper use of cryptography, one can ensure the confidentiality of data, protect data from unauthorized modification, and authenticate the source of data. Cryptography can also enable many other security goals as well. Cryptography, however, is not a panacea. Getting cryptography right is extremely hard. We list common pitfalls.

- **Rolling your own cryptographic algorithms or implementations**. Designing a cryptographic algorithm (including protocols and modes) requires significant and rare mathematical skills and training, and even trained mathematicians sometimes produce algorithms that have subtle problems. There are also numerous subtleties with implementing cryptographic algorithms. For example, the order of operations involved when exponentiating a number—something common in cryptographic operations—can leak secret information to attackers. Standard algorithms and libraries are preferable.
- **Misuse of libraries and algorithms**. Even when using strong libraries, do not assume that just using the libraries will be sufficient. There have been numerous instances in which standard libraries were used, but the developers using the libraries made incorrect assumptions about how to use the library routines. In other situations, developers don't choose the right algorithm or use the algorithm incorrectly. For example, an encryption scheme may protect the confidentiality of data, but may not protect against malicious modifications to the data. As another example, if an algorithm requires an initialization vector (IV), then choosing an IV with certain properties may be required for the algorithm to work securely. Understanding the nuances of algorithm and library usage is a core skill for applied cryptographers.

Cryptographic algorithms often don't interact nicely.

- **Poor key management**. When everything else is done correctly, the security of the cryptographic system still hinges on the protection of the cryptographic keys. Key management mistakes are common, and include hard-coding keys into software (often observed in embedded devices and application software), failure to allow for the revocation and/or rotation of keys, use of cryptographic keys that are weak (such as keys that are too short or that are predictable), and weak key distribution mechanisms.
- **Randomness that is not random**. Confusion between statistical randomness and cryptographic randomness is common. Cryptographic operations require random numbers that have strong security properties. In addition to obtaining numbers with strong cryptographic randomness properties, care must be taken not to re-use the random numbers.
- **Failure to centralize cryptography.** Numerous situations have been observed in which different teams within an organization each implemented their own cryptographic routines. Cryptographic algorithms often don't interact nicely. Best practices indicate getting it "right" once and reusing the component elsewhere.
- **Failure to allow for algorithm adaptation and evolution**. For more on this, please see "Design for changes in the security properties of components beyond your control" in the "Be Flexible When Considering Future Changes to Objects and Actors" section.

Cryptography is so hard to get right that it *always* makes sense to work with an expert if you can. Note that expertise in applied cryptography is not the same as being a mathematician and having a mathematical understanding of cryptography. At the highest level, make use of proven algorithms and libraries, but realize that *just* the use of such things does not guarantee security—it is easy to accidentally misuse these things. Have a cryptography expert work with your designers to provide an API abstraction around a strong library, so that your developers are not making decisions on algorithms and cipher modes, and so that if you need to change algorithms behind that abstraction layer, you can.

IDENTIFY SENSITIVE DATA AND HOW THEY SHOULD BE HANDLED

Data are critical to organizations and to users. One of the first tasks that systems designers must do is identify sensitive data and determine how to protect it appropriately. Many deployed systems over the years have failed to protect data appropriately. This can happen when designers fail to identify data as sensitive, or when designers do not identify all the ways in which data could be manipulated or exposed.

Data sensitivity is context-sensitive. It depends on many factors, including regulation (which is often mandatory), company policy, contractual obligations, and user expectation. Note that sensitive data are not always user-generated input. Rather, they include data computed from scratch, data coming from external sensors (for example, geolocation and accelerometer data on mobile devices), cryptographic material, and Personally Identifiable Information (PII). Creating a policy that explicitly identifies different levels of classification is the first step in handling data appropriately.

It is important to factor all relevant considerations into the design of a data sensitivity policy. For example, there are numerous regulations that system designers must consider, ultimately creating a unified approach that consistently addresses them all. A number of examples may help to flesh this out: various jurisdictions impose regulations on how personal data should be handled (such as medical records); the EU Data Protection Directive differs from the regulations in the United States; and PCI compliance issues, though not regulatory, directly affect data protection requirements.

Not all data protection requirements are the same. For some data, confidentiality is critical. Examples include financial records and corporate intellectual property. For data on which business continuity or life depends (for example, medical data), availability is critical. In other cases, integrity is most important. Spoofing or substituting data to cause a system to misbehave intentionally are examples of failures to ensure data integrity. Do not conflate confidentiality alone with data protection.

Technical data sensitivity controls that a designer might consider include access control mechanisms (including file protection mechanisms, memory protection mechanisms, and database protection mechanisms), cryptography to preserve data confidentiality or integrity, and redundancy and backups to preserve data availability.

Data sets do not exist only at rest, but in transit between components within a single system and between organizations. As data sets transit between systems, they may cross multiple trust boundaries. Identifying these boundaries and rectifying them with data protection policies is an essential design activity. Trust is just as tricky as data sensitivity, and the notion of trust enclaves is likely to dominate security conversations in the next decade.

Policy requirements and data sensitivity can change over time as the business climate evolves, as regulatory regimes change, as systems become increasingly interconnected, and as new data sources are incorporated into a system. Regularly revisiting and revising data protection policies and their design implications is essential.

ALWAYS CONSIDER THE USERS

Almost every software system in existence today interacts in one way or another with human beings. The users of a software system range from those in charge of fielding, configuring, and maintaining it operationally to those who actually use it for its intended purpose, the system's end users.

The security stance of a software system is inextricably linked to what its users do with it. It is therefore very important that all security-related mechanisms are designed in a manner that makes it easy to deploy, configure, use, and update the system securely. Remember, security is not a feature that can simply be added to a software system, but rather a property emerging from how the system was built and is operated.

The way each user interacts with software is dictated not only by the design and implementation decisions of its creators but also by the cognitive abilities and cultural background of its users. Consequently, it is important that software designers and architects consider how the physical abilities, cultural biases, habits, and idiosyncrasies of the intended users of the system will impact its overall security stance. It is also a truism that during the life of any moderately useful system, a few users will discover capabilities that are outside the intentions of the system's designers and builders. Some of those capabilities may very well have significant security implications.

Usability and user experience considerations are often the most important factors ensuring that software systems operate in a secure manner. Designing systems that can be configured and used in a secure manner with easy-to-use, intuitive interfaces and sufficiently expressive, but not excessive, security controls is crucial.

However, it is dangerous to assume that every intended user of the system will be interested in security—or will even be well-meaning. The challenge to designers and architects lies in creating designs that facilitate secure configuration and use by those interested in doing so, designs that motivate and incentivize secure use among those not particularly interested in software security, and designs that prevent or mitigate abuse from those who intend to weaken or compromise the system.

Failing to address this design principle can lead to a number of problems:

- Privilege escalation may result from a failure to implement an authorization model that is sufficiently tied to the authenticated entity (user) in all cases. Escalation failures may also occur when higher-privileged functions are not protected by the authorization model and where assumptions about inaccessibility are incorrect.
- A particular failure of appropriate authorization can allow a breach of the intended authorization and isolation between users

such that one user may access another user's data.

- When designers don't "remember the user" in their software design, inadvertent disclosures by the user may take place. If it is difficult to understand the authorization model, or difficult to understand the configuration for visibility of data, then the user's data are likely to be unintentionally disclosed.
- Default configurations that are "open" (that is, default configurations that allow access to the system or data while the system is being configured or on the first run) assume that the first user is sophisticated enough to understand that other protections must be in place while the system is configured. Assumptions about the sophistication or security knowledge of users are bound to be incorrect some percentage of the time. This is particularly true at the startup and initialization of the system.
- If the security configuration is difficult or non-intuitive, the result will be an inability to configure the product to conform to the required security policy.
- Designers sometimes fail to account for the fact that authenticated and properly authorized users can also be attackers! This design error is a failure to distrust the user, resulting in authorized users having opportunities to misuse the system.
- When security is too hard to set up for a large population of the system's users, it will never be configured, or it will not be configured properly. This is especially dangerous where the system's defaults are "open" or insecure. For example, if there are too many clicks required for the user to get from the main page or screen to a security control panel, users are unlikely to persist through the labyrinth of clicks.
- Failure to consider the needs of programmers who must code to an API will cause the intended automation patterns to be missed. Programmers are a class of users who also require that the interface they consume be intuitive enough to guide them to correct usage patterns. Because a misunderstanding of an API occurs within the program that uses it, problems may not be readily apparent (appearing perhaps only obliquely, within log files of the ongoing activity), and the debugging of the problem difficult; this failure can be one of the most difficult to find and fix. Additionally, if the API must be changed, many if not all consumers of the API may be forced into further changes, thus spreading the original failure throughout the ecosystem.
- Failure to consider the possibility of "collateral damage" that can occur from included or embedded software or data in the user interface may cause an inadvertent or unintentional leaks of personal data. Consider the capture of a bystander in a personal photo taken in a public place. Even if that passerby is not a user of software, the bystander's privacy may be compromised if that image is posted online later.
- Failure to consider the user's data during setup, use, and revocation/termination may cause unintended data to be gathered and stored against the users' wishes, or may hold onto data that should have been removed completely after the user has stopped using the service and closed his or her account. For example, when a user decides to stop using the system, is the private data easy for the user to destroy?
- Failure to consider the many different classes of users (blind users, language proficiency, children, people with different mental

capabilities, etc.) will exclude those classes of users from the software -- or, alternatively, make the software too difficult to use effectively. Most importantly, when designing the security of the system, failure to consider how security is set up and used from the perspective of users with different capabilities and understandings typically causes those users to set up and make inappropriate use of the software's security.

Stepping back, our biggest recommendation is the following: Always consider the users, and any other stakeholders, in the design and evaluation of systems. There are numerous factors to consider, and there are often trade-offs; for example, improving the system with respect to one user value (such as privacy or usability) can negatively affect another user value (like ease of accessing the relevant information).

In addition to the general recommendations given above, there are numerous artifacts designers can consider in order to address specific problems mentioned earlier. The decision whether to implement these specific recommendations will, however, depend on the system in question. For example, in some cases we recommend not putting security-relevant decisions in the hands of all users, as they may not possess the knowledge or context to evaluate those decisions. Similarly, because users may not know how to explore or choose between a variety of options, we recommend making the easiest and most common usage scenario also secure—a notion often referred to as "secure by default." When users do desire to change security settings, we suggest making it as easy as possible for them to find the relevant settings.

Often there is value in allowing users to test different security and privacy settings and see the results in order to understand the impact of the changes (for example, on social networks, good interfaces allow users to see their privacy-settings changes from the perspective of other users).

On the other hand, it might be preferable not to give the user a choice at all; or example if a default secure choice does not have any material disadvantage over any other; if the choice is in a domain that the user is unlikely to be able to reason about; or if one user's choice may significantly affect the system's or the other user's state, including security.

Designers must also consider the implications of user fatigue (for example, the implications of having a user click "OK" every time an application needs a specific permission) and try to design a system that avoids user fatigue while also providing the desired level of security and privacy to the user.

The field of user-focused security is rich with tensions. As a trivial example, so-called "secure" password selection strategies are also well known to lead to passwords that are hard for users to remember. A more complex example of these inherent tensions would be the need to make security simple enough for typical users while also giving sophisticated or administrative users the control that they require. We encourage designers to also consider other resources on designing security systems with stakeholders in mind.

By fully considering all the relevant stakeholders, designers have the opportunity to create systems that are both secure and usable, systems that will see adoption, and systems that will be compatible with the values of users and other people impacted by them.

UNDERSTAND HOW INTEGRATING EXTERNAL COMPONENTS CHANGES YOUR ATTACK SURFACE

It is unlikely that you will develop a new system without using external pieces of software. In fact, when adding functionality to an existing system, developers often make use of existing components to provide some or all of that new functionality. In this context, external components refer to software "not written here," such as:

- Software procured as off-the-shelf components, platforms, and applications
- Third-party open source or proprietary libraries
- Widgets and gadgets added or loaded at runtime as part of a web project
- Access to some functionality provided by the component that you plan to take advantage of (such as accessing a web service that provides federated authentication)
- Software developed by a different team within your organization
- Software that your team developed at a previous point in time (perhaps at a time when the security stance was not as mature as it is now)

These components may be included as binaries, libraries, and source code, or they may exist simply as APIs.

It is a common adage of software security that whenever possible, functionality should be achieved by the reuse of tried-and-true pieces of previously tested and validated software, instead of developing from scratch every time. The important distinction is that the software being newly included has actually been tried as well as tested and found to stand up to your current standards of software security. The decision to use-rather-than-build means that the software as a whole inherits the security weaknesses, security limitations, maintenance responsibility, and the threat model of whatever you are including. This inheritance can amount to a deficit of security, which must be solved, mitigated, or accounted for when the system is finished. The system's "threat model" is a representation of the security posture of the system when all possible threats are taken into consideration, their mitigations established, and the vulnerabilities identified.

Make sure you allocate time in your software development methodology to consider the security impact on your system when including an external component:

- How does the external component change the threat model of the entire system? Does it add to the attack surface? Does it modify entry points in the system that had already been considered in its own threat model?
- Were new features, capabilities, or interfaces added even though you are not using them? Can those unused features be disabled?
- Does the external component being included also include other external components with their own security weaknesses?
- Have you obtained the external component from a known, trusted source?
- Does the external component provide security documentation that may help you better understand its threat model and the security implications of its configuration?

You must assume that incoming external components are not to be trusted until appropriate security controls have been applied, in order to align the component's attack surface and security policy with ones that meet your requirements.

Examples of potential security issues with third-party components include the following:

- Loading a library with known vulnerabilities (CWE, CVE, etc.)
- Including a library with extra features that entail security risks
- Reusing a library—yours or a third party's—that no longer meets current software security standards
- Using a third-party service and hoping thereby to pass responsibility of security to that service
- Configuration mistakes in the security of a library—e.g, secure defaults
- Library making outbound requests to the maker's site or to some partner of theirs
- Library receiving inbound requests from some external source
- A single external component including other components, causing multiple levels of inclusion ("recursion")
- Including pieces of functionality that offer unknown interfaces into the system—for example, a CLI for configuration of an included daemon, a panel or admin mode for a Web component, a hardcoded set of credentials for an authentication/authorization module, a debugging interface or backdoor, or the like.

At a minimum, consider the following:

- Isolate external components as much as your required functionality permits; use containers, sandboxes, and drop privileges before entering uncontrolled code.
- When possible, configure external components to enable only the functionality you intend to use.
- If you include functionality that you do not intend to use, you must consider how that included functionality changes your security posture (attack surface, inherited debt, threats, etc.), and therefore increases the security you must implement to account for the change.
- If you cannot configure the security properties of the component to align with your security goals, find another library, or document that you are accepting the risk and inform relevant stakeholders of your decision.
- Likewise, if the element to be included cannot realize your security objectives, find a different element, or document that you are accepting the risk and inform relevant stakeholders of your decision.
- Validate the provenance and integrity of the external component by means of

cryptographically trusted hashes and signatures, code signing artifacts, and verification of the downloaded source. If no integrity mechanism is available, consider maintaining a local mirror of the library's source. Understand the risk of dynamically including components such as JavaScript from external sources. If the external host is compromised you may be including attacker-controlled JavaScript.

- Identify and follow sources that track or publish security-related information regarding the external components you consume: bug repositories, security-focused mailing lists, CVE databases, and so forth.
- Make sure that the development team members charged with responding to security events are aware of all external components used so those can be included in their threat intelligence collection efforts.
- Maintain an up-to-date list of consumed external components and at a pre-established cadence verify that it matches the versions included in your product, as well as that those are the latest known-secure versions available for each external component.
- Maintain a healthy distrust of external components:
- Whenever possible, authenticate the dataflow between your system and external components.
- Consider all data coming from an external component to be tainted, until proven valid (see "Define an approach that ensures all data are explicitly validated" for additional information).
- Be sure to understand and verify the default configuration of the external component. For example, if you are including an external crypto library, understand what values are used by default unless you change them; for example, sources of entropy, algorithms, and key lengths. When consuming an external component such as a Web server, understand its defaults concerning admin modes, ports where the processes will be listening, and assumptions concerning how it interfaces with the operating system and with your own software.
- Document everything. If you change a default, make sure that there is documentation as to why the decision was made to change it. If you include an external component, create documentation around the process used to choose the component, the provenance of the component, the verification it went through, and most importantly any security-relevant assumption made about it. This will make it easier to move forward when versions change, or when you consider the use of an alternative external component. When changing the build defaults of external components, configuration options for deployment, or source code, automate the procedure using your version control system or a patch file (numerous tools, including make, sed, and patch, are available for this task depending on your environment). Then include the automated procedure in your build workflow—bring in the pristine component, apply your modifications, and use it for your build. The automation will help to maintain consistency between builds, and some tools include calling modes or executables that validate their own configurations; leverage those into your process as well to know when your modifications need adjustment due to a version change in the external component or some other similar event.
- Design for flexibility. Sometimes an external component becomes too risky, or its development is abandoned, or the functionality it offers is surpassed by another external component. For those cases, you will want to design your system so that external components can be easily replaced.

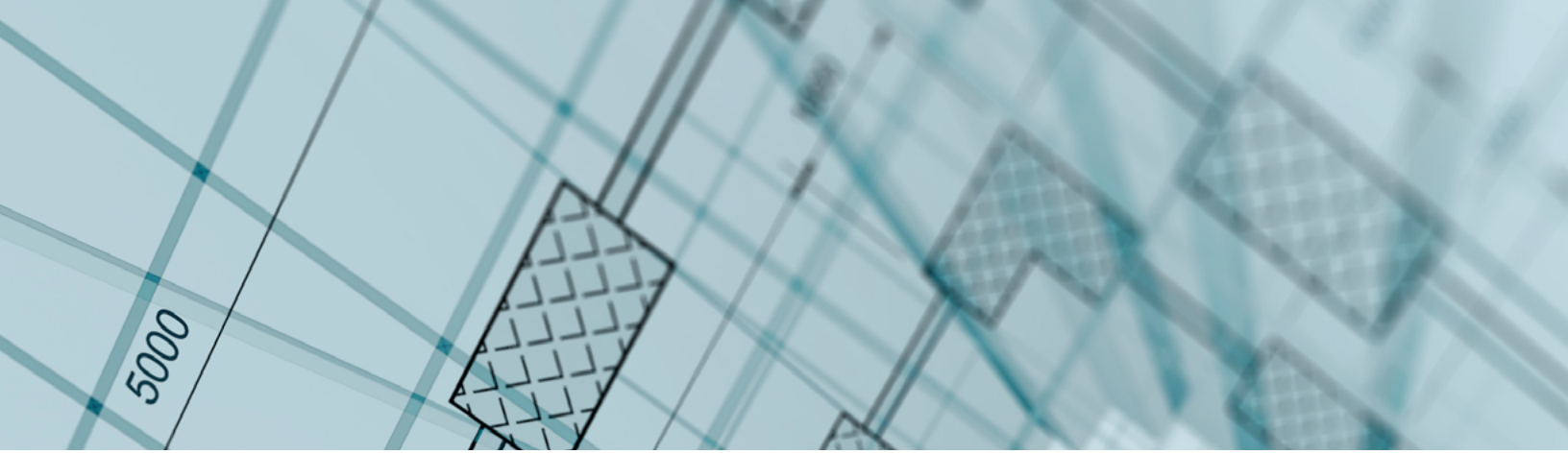
BE FLEXIBLE WHEN CONSIDERING FUTURE CHANGES TO OBJECTS AND ACTORS

Software security must be designed for change, rather than being fragile, brittle, and static. During the design and development processes, the goal is to meet a set of functional and security requirements. However, software, the environments running software, and threats and attacks against software all change over time. Even when security is considered during design, or a framework being used was built correctly to permit runtime changes in a controlled and secure manner, designers still need to consider the security implications of future changes to objects and actors.

Designers need to understand how change influences security considerations under many circumstances. There will be changes at runtime, in the form of configuration changes, enabling and disabling of features, and sometimes dynamic loading of objects. The need for security consideration will appear during testing, since all possible variations of states will need to be verified to guarantee that they uphold the security posture of the system (among, of course, other tested behavior). There will be changes at deployment when permissions, access control and other security-related activities and decisions need to take place. The addition of continuous integration processes creates a requirement for security flexibility, as changes to systems are pushed automatically and at ever shorter periodicity.

Meanwhile, entropy increases in every way possible. Threats change over time. Embedded components (that is, components that are not easily reachable) will inevitably be found to be vulnerable to attacks, researchers will discover new ways to break into systems, and proprietary code will reveal itself to contain vulnerabilities. Any deployed system can eventually come under attack and potentially be compromised. And, because threats change over time, even a deployed system that has resisted attacks for a long time may eventually succumb to an attack and be compromised.

Like threats, the environment and conditions under which the system exists will also change. It is a different proposition to maintain security for a system with 10 users than 10 million users —not at all a simple matter of linear scale. A system that works well in a given configuration might find itself exposed to new threats by virtue of changes to that environment; for



example, the addition of a mobile interface to a legacy system.

For these reasons, secure design keeps flexibility in mind.

Design for secure updates. It is easier to upgrade small pieces of a system than huge blobs. Doing so ensures that the security implications of the upgrade are well understood and controlled. For example, a database engine upgrade may involve new access control defaults or rewrites of the controls such that previously tight permissions loosen, or create new default users that need to be disabled. If the update happens with the same change operation performed on the web server, the amount of change and adjustment to a dynamic, already-configured system may be overwhelming to track and assure.

Have the system being upgraded verify the integrity and provenance of upgrade packages; make use of code signing and signed manifests to ensure that the system only consumes patches and updates of trusted origin. This is a non-trivial design consideration, as there are many details in process and implementation that may break if poorly thought out beforehand. Finally, consider the maintenance burden placed on administrative personnel. As complexity increases, there is an increasing likelihood of making mistakes.

Design for security properties changing over time; for example, when code is updated. If the system ran in a small environment yesterday, and local users and password storage were sufficient, tomorrow the system may be changed to make use of an alternate identity management solution. In that case, the migration of previous users (and/or the correct coexistence of the local and remote users) would need to happen in a way that does not compromise security; for example, there should be consideration of user ID collisions such as when a remote and a local user have the same username.

Design with the ability to isolate or toggle functionality. It should be possible to turn off compromised parts of the system, or to turn on performance-affecting mitigations, should the need arise. Not every vulnerability identified can be readily mitigated within a safe time period, and mission-critical systems cannot simply be taken offline until their vulnerabilities are addressed. For example, in certain environments a stateful firewall may impact performance overall, and so it is turned off – until a vulnerability that may be stopped by turning it on is identified, in which case it becomes worthwhile to bear the performance cost by turning the firewall on until a proper patch can be developed, tested, and applied.

Design for changes to objects intended to be kept secret. History has shown us that secrets such as encryption keys and passwords get compromised. Keeping secrets safe is a hard problem, and one should be prepared to have secrets replaced at any time and at all levels of the system. This includes several aspects:

And because threats change over time, even a deployed system that has resisted attacks for a long time may eventually succumb to an attack and be compromised.

- A secure way for users to change their own passwords, including disallowing the change until the old password has been successfully presented by the user.
- Carefully considering any kind of "password recovery" mechanism. It is better to give the forgetful user a way to reset their password after verification via a parallel mechanism (like email) than to provide the password in clear text, which can be subverted or compromised in any number of ways.
- A secure and efficient way to replace certificates, SSH keys, and other keys or authentication material that systems use, providing clear and explicit logs of those events (without including the secrets in any form!) in a forensically verifiable way (for example, external log servers and checksums).
- Understanding how the key change affects data stored at rest. For example, if data are encrypted on a file system or in a database and an administrator needs to change the encryption key, is it better to decrypt all data using the current key and re-encrypt that data with the new key, or to maintain versions of encrypted data and encryption keys?

Design for changes in the security properties of components beyond your control. Tech marches on. A cipher that was considered secure yesterday may be found to be less secure today, either by the discovery of an active attack against it or by improvements in hardware and software able to defeat that security control. In the same way, an external component's security properties or related characteristics may change over time, as when an Open Source project is abandoned and its code not actively maintained, or when its license changes, forcing users to abandon it.

In these cases it is important to design "agility," the capability to change layers and algorithms as needed, into the system. Good examples include Java's capability to change crypto providers without recompilation of classes, and Apache's capability of specifying a list of ciphers it is willing to negotiate with a client. Many hours of development and much grief over security flaws have been avoided due to these capabilities. Good design allows for intermediate layers of abstraction between code and imported external APIs, so that developers can change components providing needed functionality without changing much of the code.

Design for changes to entitlements. Systems are sometimes designed in which support staffers have privileged access to certain parts of the system in order to perform their job. However, the support staff's access to various system components likely changes over time. Individuals leave the organization, job functions change, they go on extended leaves or sabbaticals, system functionality changes, and so on. The system must have a way to revoke access to areas when a user no longer has a need to access them. This revocation of access should be part of an existing auditing mechanism in which access to critical system components is regularly reviewed to confirm that those individuals with access still require that level of access.

GET INVOLVED

As stated in the mission statement, the IEEE Computer Society Center for Secure Design will provide guidance on:

- Recognizing software system designs that are likely vulnerable to compromise.
- Designing and building software systems with strong, identifiable security properties.

This document is just one of the practical artifacts that the Center for Secure Design will deliver.

IEEE CENTER FOR SECURE DESIGN

Interested in keeping up with Center for Secure Design activities? Follow @ieeecsd on Twitter, catch up with us via cybersecurity.ieee.org, or contact Kathy Clark-Fisher, Manager, New Initiative Development (kclark-fisher@computer.org).

About IEEE Computer Society

IEEE Computer Society is the world's leading computing membership organization and the trusted information and career-development source for a global workforce of technology leaders. The Computer Society provides a wide range of forums for top minds to come together, including technical conferences, publications, and a comprehensive digital library, unique training webinars, professional training, and the TechLeader Training Partner Program to help organizations increase their staff's technical knowledge and expertise. To find out more about the community for technology leaders, visit http://www.computer.org.

IEEE computer society


5000
5000