Application Software Security and the CIS Controls

A Reference Paper

Authors:
Steve Lipner, SAFECode
Stacy Simpson, SAFECode

01010
0100000
010001
01000
0100001
SAFECode
Driving security and integrity.

Introduction

The CIS Controls®[1] is a prioritized set of cyber security practices that are targeted toward defending against today's most pervasive attacks. The CIS Controls are not a catalog of every best cyber defense practice, but rather a list of high-value actions that provide organizations at all maturity levels with a starting point of "must do" activities to optimize their cybersecurity programs.

Led by the Center for Internet Security® (CIS®), the CIS Controls have matured into an international community of volunteer individuals and institutions that:

- Share insights into attacks and attackers, identify root causes, and translate that into classes of defensive action
- Create and share tools, working aids, and stories of adoption and problem-solving
- Map the CIS Controls to regulatory and compliance frameworks in order to ensure alignment and bring collective priority and focus to them
- Identify common problems and barriers (like initial assessment and implementation roadmaps), and solve them as a community

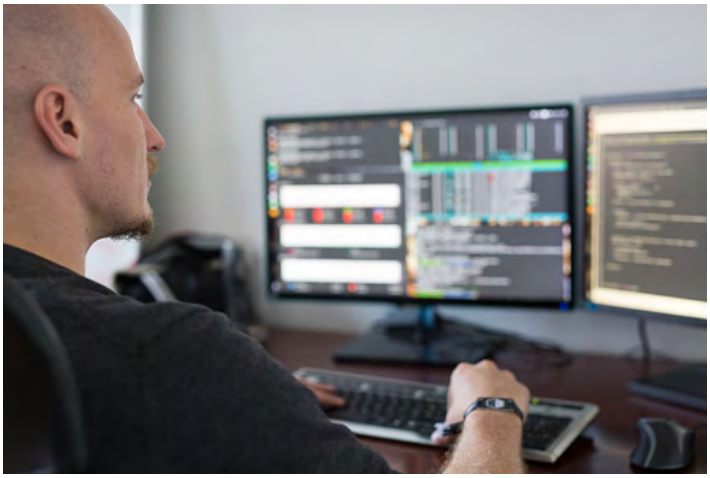
Application software security is a relatively recent addition to the set of controls. Once a discipline that was thought to be highly specialized and largely a responsibility limited to software engineers, software security has grown into a core focus for organizations that have come to recognize the need to manage the security risk of both in-house developed and acquired software. This means many organizations must now have an understanding of software security practices as both buyers and creators of software.

There is a long history of work in software security and secure software development, starting with Microsoft's Security Development Lifecycle,[2] which influenced much of the effort to define a common process for reducing vulnerabilities in developed software. Industry groups such as SAFECode[3] and the Open Web Application Security Project (OWASP)[4] continue to work to advance and promote software security best practices. The Software Alliance, also known as BSA, published the BSA Framework for Security Software[5] to provide a common language to help suppliers, customers and policymakers better describe and evaluate application software security. Most recently, the National Institute of Standards and Technology (NIST) brought together much of what has been learned over the past two decades and published "Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework (SSDF)[6]." NIST's framework reinforces that secure software is a result of a holistic software security process and offers guidance as to what constitutes an effective process, incorporating recommendations from SAFECode, BSA, OWASP, and others. In some ways, it can be thought of as the application software security version of the NIST Cybersecurity Framework[7].

So why develop yet another paper? Much of today's most recognized software security guidance was written as a catalog of practices or targeted toward large software development organizations. Viewing software security in the context of the CIS controls requires us to think a bit differently about our approach. It asks us to consider how we would prioritize application software security practices for organizations just starting out or with limited resources. It also asks us to consider how to help those who run broader cybersecurity programs better assess and manage their risk from in-house and acquired software. Control 16 in the CIS Controls was developed from this dual perspective. Bringing all of these considerations together in a single control was not an easy task. Thus, we opted to further document the thinking behind the control's development and create this reference paper to further support those seeking a deeper understanding of application software security.

1 https://www.cisecurity.org/controls/v8/
2 https://www.microsoft.com/en-us/securityengineering/sdl
3 www.safecode.org
4 https://owasp.org/
5 https://www.bsa.org/reports/updated-bsa-framework-for-secure-software
6 https://csrc.nist.gov/News/2020/mitigating-risk-of-software-vulns-ssdf
7 https://www.nist.gov/cyberframework

Purpose and Methodology



This paper will complement Control 16, Software Security, of the CIS Controls by providing expanded guidance to help organizations in their efforts to implement the control. Though aligned with existing works like SAFECode's Fundamental Practices and the NIST Software Security Framework, this paper will be more prescriptive than a catalog of recommended practices. Like the CIS Controls, It aims to prioritize recommended software security controls by implementation group and maturity level.

This paper is not meant to replace existing software security frameworks. Rather, it can be read as a starting point for organizations looking to implement software security programs, allowing them to direct their limited resources at high value activities. More mature organizations or those with additional resources should consider these recommended practices foundational and will likely have broader programs with additional activities that best address their unique business risks.

All recommendations made by SAFECode are derived both from industry-accepted best practices and its members' real-world experiences in creating, managing and executing software security programs. As a result, the guidance is both practical and tested in diverse software development environments. Primary industry references will include published guidance from SAFECode, such as its Fundamental Practices for Secure Software Development[8] and Managing Security Risks Inherent in the Use of Third Party Components[9] papers; the NIST SSDF[10]; The BSA Framework for Secure Software,[11] and work from OWASP.[12]

Audience

This document is primarily focused on organizations that do development, but it will also provide related considerations for buyers of software who are seeking to understand whether their vendors are using these practices as part of their efforts to manage the risk of acquired software.

Fully addressing the needs of those looking to manage risk from acquired software is beyond the scope of this paper. It is also an area of work that has continued to challenge both vendors and customers. While customers have a need to assess the security of acquired software, there has been no widely accepted industry standard to assess the security of software. This has created challenges not only for customers, but also for suppliers who have a responsibility to effectively communicate risk. Fortunately, the industry is beginning to coalesce around a process-based approach to evaluating software security. The release of the NIST SSDF represents a big step forward in creating a framework for the assessment of secure development processes. Other efforts, like the in-progress development of ISO 27034 and IEC 62443 (focused on industrial cybersecurity), are also working toward supporting this goal.

8 https://safecode.org/fundamental-practices-secure-software-development/
9 https://safecode.org/managing-security-risks-inherent-in-the-use-of-third-party-components/
10 https://csrc.nist.gov/News/2020/mitigating-risk-of-software-vulns-ssdf
11 https://www.bsa.org/reports/updated-bsa-framework-for-secure-software
12 https://owasp.org/

However, as the industry works toward this longer-term goal, we recognize the pressing need of security professionals to manage their risk from both in-house and acquired software. As such, we seek to include buyers in the conversation by keeping the paper focused and accessible to readers without a deep software engineering background, and by including a discussion of the principles on which to base an effective vendor assessment.

How to Use this Paper

This paper outlines a prioritized list of high-value security practices that should be a part of nearly all software security programs. For organizations just starting out or with limited resources, these practices provide an effective starting point for addressing software security. For more mature organizations, the effective execution of these practices should be the foundation of your secure development program. This doesn't mean there won't be exceptions to the order of activities listed based on an individual risk profile or even that there won't be additional activities outside of these listed that are high value. Finally, readers of the recommendations will note the primary role that root cause analysis plays in this approach. Put simply, root cause analysis allows organizations to identify the security issues that most frequently or seriously impact their codebase. This knowledge can help all organizations further refine the prioritization of their efforts.

Software development today is very different from the usual practice of fifteen or twenty years ago. This paper provides guidance that is relevant to today's changed development challenges:

- First, much development is based on open source, and many developers of products or services rely heavily on open source components that other individuals or organizations created. This document directly addresses the issues associated with securely including open source or third-party components in software.
- Second, the development paradigm has changed from a waterfall or spiral lifecycle that produced a software release in months or years to Agile or DevOps lifecycles that release software in weeks, days, or even hours or minutes. The practices identified in this document are lifecycle-independent: individual tools, techniques, and training can and should be integrated into disciplined software development processes no matter whether they are waterfall, DevOps, or something in between.
- Finally, software deployment has changed from a world of packaged software that customers deployed on-premises in their own environments to a world where applications run on remote servers "in the cloud." While the transition to cloud-hosted software changes the environment, the fundamental considerations such as building secure software, relying on platform security features, and minimizing attack surface - as introduced in this document - remain unchanged.

The recommended practices are roughly broken into three groups based on maturity level, which are aligned with the Implementation Groups in the CIS Controls. Software security is largely an Implementation Group 2 and 3 focus area, a reflection of not just its complexity, but also of the idea that many of the types of organizations envisioned as part of CIS Implementation Group 1 do not typically do software development, rather relying on commercial off-the-shelf or open source solutions in small or home business environments.

In thinking about how to align our recommended practices to these Implementation groups, we felt the diversity of organizations within the software development ecosystem created a need to supplement the CIS Implementation

Groups with more targeted software security classification levels to increase the usefulness of this reference paper to readers. As such, we've created three tiers for the recommended practices that loosely reflect the complexity of the development environment. An organization may find that its CIS Implementation Group and its Development Group as defined below will differ - that is, an Implementation Group 2 organization could find that its software security needs align with Development Group level 1, 2, or 3.

Development Group 1

- The organization largely relies on off-the-shelf or open source (OSS) software and packages with only the occasional addition of small applications or website coding. The organization is capable of applying basic operational and procedural best practices and of managing the security of its vendor-supplied software by following the guidance of the CIS Controls.

Development Group 2

- The organization relies on some custom (in-house or contractor-developed) web and/or native code applications integrated with third-party components and running on-premises or in the cloud. The organization has a development staff that applies software development best practices. The organization is attentive to the quality and maintenance of third party open source or commercial code on which it depends.

Development Group 3

- The organization makes a major investment in custom software that it requires to run its business and serve its customers. It may host software on its own systems, in the cloud, or both and may integrate a large range of third-party open source and commercial software components. Software vendors and organizations that deliver software as a service should consider Maturity Level 3 as a minimum set of requirements, but may well have to exceed those requirements in some areas.

Generally speaking, organizations that self-define as Development Group 2 will be doing practices described under Development Groups 1 and 2 and those that self-define as Development Group 3 will be doing practices described under Development Groups 1, 2 and 3.

Implementation Groups

IG1

An IG1 enterprise is small to medium-sized with limited IT and cybersecurity expertise to dedicate towards protecting IT assets and personnel. The principal concern of these enterprises is to keep the business operational, as they have a limited tolerance for downtime. The sensitivity of the data that they are trying to protect is low and principally surrounds employee and financial information.

Safeguards selected for IG1 should be implementable with limited cybersecurity expertise and aimed to thwart general, non-targeted attacks. These Safeguards will also typically be designed to work in conjunction with small or home office commercial off-the-shelf (COTS) hardware and software.

IG2 (Includes IG1)

An IG2 enterprise employs individuals responsible for managing and protecting IT infrastructure. These enterprises support multiple departments with differing risk profiles based on job function and mission. Small enterprise units may have regulatory compliance burdens. IG2 enterprises often store and process sensitive client or enterprise information and can withstand short interruptions of service. A major concern is loss of public confidence if a breach occurs.

Safeguards selected for IG2 help security teams cope with increased operational complexity. Some Safeguards will depend on enterprise-grade technology and specialized expertise to properly install and configure.

IG3 (Includes IG1 and IG2)

An IG3 enterprise employs security experts that specialize in the different facets of cybersecurity (e.g., risk management, penetration testing, application security). IG3 assets and data contain sensitive information or functions that are subject to regulatory and compliance oversight. An IG3 enterprise must address availability of services and the confidentiality and integrity of sensitive data. Successful attacks can cause significant harm to the public welfare.

Safeguards selected for IG3 must abate targeted attacks from a sophisticated adversary and reduce the impact of zero-day attacks.

To use this reference paper, identify your Development Group and let that guide you in prioritizing your investments and resources. Organizations new to software security can use this paper to build a program while more mature organizations can reflect on whether they are missing anything or need to strengthen certain areas. Keep in mind that this is meant as a rough guide for prioritization; without knowing your environment, we can't prioritize for you but we can provide guidance to help you do so based on broadly accepted industry practices and direct experience in managing the security aspects of software development programs. You will have to make final decisions on tools and techniques as a function of your platform (*e.g.*, Linux vs Windows; cloud vs on-premises) and technology (*e.g.*, JavaScript web applications vs native C++ code). You'll also note the high prioritization of root cause analysis, which should play a central role in helping you develop your security program. The tiered approach is also designed to help readers develop a future roadmap and understand how software security will evolve alongside their organizational growth.

Recommended Practices: Development Group 1 / CIS Implementation Group 2

The following recommendations provide a reasonable starting point for organizations new to software security. They encompass activities largely aimed at helping organizations create a process-based approach to developing a more rigorous understanding of their unique risk profile and the security events they most commonly face. These recommendations will also help develop a basic determination of how their risk level aligns with security activities they currently perform.

While these recommendations are targeted at those with new programs, more mature organizations will recognize the need to continually focus on and improve these practices as their effective execution is absolutely critical to the ongoing success of their broader software security efforts.

Create and Manage a Vulnerability Response Process

It may seem strange to start with vulnerability response when the goal of software security is preventing vulnerabilities in the first place. Yet no matter how mature and effective a software security program may be, the complexity of software development and an ever-changing threat environment means there is always potential for vulnerabilities to exist in released software that may be exploited to compromise the software or its data. Therefore, the vulnerability response and disclosure process should be viewed as an integral element to any software security program. Such a process not only should help drive the resolution of externally discovered vulnerabilities with customers and stakeholders, but as we'll discuss in the next section, it should also inform other elements of a software security program. The goal of the process is to provide customers with timely information, guidance, and mitigations or updates to address their risks from discovered vulnerabilities.

Detailed guidance on the creation and management of a vulnerability response program can be found in two related ISO Standards, ISO/IEC 29147[13] and ISO/IEC 30111[14], and at groups like FIRST.[15] Here we will discuss the simplest, most common steps organizations should be able to take regardless of maturity level.

- **Be prepared to accept vulnerability reports:** Create an internal vulnerability handling policy that defines who is responsible in each stage of the vulnerability handling process and how they should handle information on potential and confirmed vulnerabilities. In large organizations, this may involve setting up a Product Security Incident Response Team (PSIRT). In smaller organizations, this could be a single individual or a small group of secure development team members. It is important that someone is responsible and accountable for the timeline of handling vulnerability reports and that there is a clear process for them to follow when a report is received. Ideally, organizations would also create an externally-facing policy that helps to set expectations for outside stakeholders on what they can expect when a potential vulnerability is found.

13 ISO/IEC 29147 – Vulnerability disclosure. Provides a guideline on receiving information about potential vulnerabilities from external individuals or organizations and distributing vulnerability resolution information to affected customers

14 ISO/IEC 30111 - Vulnerability handling. Gives guidance on how to internally process and resolve reports of potential vulnerabilities within an organization

15 https://www.first.org/

- **Make it clear how to report a vulnerability and respond to reports:** Ensure that vulnerability reporters or security researchers, customers or other stakeholders know where, how, and to whom to report the vulnerability. Ideally, this intake location is easily discoverable on your company website along with a dedicated email alias. If you have a reporting process for security researchers that is different from the process for customers, such information should be clearly called out. Since information about vulnerabilities may be used to attack vulnerable products, you should provide a way for sensitive vulnerability information to be communicated confidentially if possible. Respond to all received reports using the vulnerability handling policies described above.
- **Fix reported and related vulnerabilities in a timely manner:** A reported vulnerability should be quickly triaged by the team that owns the vulnerable code to validate it, prioritize it by severity and, finally, remediate it. Once validated, things like the potential impact, likelihood of exploitation, and the scope of affected customers should be used to help determine the urgency for a fix. It is important to assess whether the reported vulnerability is a "one-off" or an instance of a problem that is repeated in multiple places in the code. In the latter case, the team should fix all instances as quickly as feasible to avoid having to deal with a flurry of related reports as security researchers dig into the affected code. It is also important to keep in touch with the individual or organization that reported the vulnerability to advise them of plans and progress and to convey to them that there's no need to "encourage a fix" by making negative public comments about your organization or software. (See the ISO standards and the FIRST[16] website for more information on communicating with vulnerability researchers.) Once a fix (or fixes) is developed, be sure to have the development team test it along with any additional mitigation advice such as configuration changes.
- **Consider clear and actionable communication to impacted customers a key part of remediation.** Open and transparent communication about identified and (hopefully) fixed vulnerabilities is a key component of a vulnerability response program. It usually includes using a common identifier for each reported vulnerability, such as an entry in the National Vulnerability Database[17] of Common Vulnerabilities and Exposures (CVE) entries, as well as assigning a criticality to help users decide how quickly they should apply the patch. The Common Vulnerability Scoring System (CVSS)[18] has been designed to quantitatively represent characteristics of a vulnerability. Finally, establish a publically available web page where users and customers can always find the latest vulnerability and patch information. This could also include an automatic notification process for consumers to proactively inform them of newly available patches or software versions.

Perform Root Cause Analysis

In the previous section, we noted that some may find it odd to start a paper on creating a software security program with vulnerability response given the focus of software security is on preventing vulnerabilities in the first place. In fact, in many diagrams and papers, vulnerability response is listed as the final stage of secure development, unintentionally leaving the impression that it is about responsibly cleaning up what you missed. But arguably, the vulnerability response process should be thought of as the first step in building a software security program because it enables you to perform root cause analysis on discovered vulnerabilities. And this is critical because root cause analysis can help those new to software security or with limited resources to knowledgeably prioritize how they approach secure software development.

Performing root cause analysis means that in addition to fixing the reported issue, the software development team should consider the nature of the defect. Is this something that is occurring over and over again? If so, then there is a need to prioritize the aspect of our software security program that specifically targets this type of flaw. Do we need to add tools or update our developer training? Root cause analysis allows you to move beyond just fixing the

16 https://www.first.org/
17 https://nvd.nist.gov/
18 https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

vulnerabilities as they arise. It enables you to identify (and eventually fix) the things in your development process that create the vulnerabilities in the first place.

By performing root cause analysis and documenting trends over time, you can tailor your software security approach to target the issues most likely to impact the security of your product. This is particularly important for smaller organizations or those with a need to prioritize resource commitments. It can also help newer software security programs to have a more immediate positive impact, thereby helping to build a business case for continued investment.

Secure Third Party Code

Today, nearly all software relies on third-party components that were created by an individual or organization outside the development team. These components can be commercial off-the-shelf-products (COTS) or open source software (OSS) projects and their scope can range from the embedding of an entire product to just using a few snippets of code. In all cases, the use of third-party components introduces potential software security risks that must be managed.

Software products inherit the security vulnerabilities of the components they incorporate so their security must be addressed. Using SAFECode's paper, *Managing Security Risks Inherent in the Use of Third-party Components*[19], there are five key phases to managing third party component risk: select, maintain, assess, mitigate, and monitor.

Select: Choose established and proven frameworks and libraries that provide adequate security for your use cases, and which defend against identified threats. For commercial products and open source distributions, the NIST-managed National Vulnerability Database[20] will help you to understand products' vulnerability track records and developers' responses, though it's important to note that widely-used and well-secured software may have more *reported* vulnerabilities than poor quality software with no security team but no users. For open source software, it's important to choose components or distributions that are under active development and maintenance and have a track record of fixing reported vulnerabilities.

Maintain: Keep an updated inventory of third party components in use, often referred to as a "bill of materials," as well as components slated for future use. For new code, this can be as simple as manually keeping a list of third-party components and their versions. For legacy code, especially large codebases where third party components have been integrated without being tracked, it may be a challenge to identify third party components, but creating this inventory is critical to future security. There are a number of ways to identify and maintain an inventory of third party components. Smaller organizations may do it manually, while larger organizations or organizations with larger codebases can use software composition analysis (SCA) tools to identify embedded third-party components. The most mature organizations tend to use a mix of SCA tools and manual inventory maintenance.

Assess: Using the inventory of third party components, assess the risk that each third-party component (TPC) poses to the software. Start with determining known security vulnerabilities and their impact. But also be sure to consider the potential for unknown issues, by reviewing the maturity of TPC's provider, such as maintenance cadence, stability of the TPC over time, development practices employed by the TPC provider, and whether the TPC will reach end of life within the expected lifetime of a product. The outcome of this step can be a risk score for a TPC of interest. This score could be binary -- acceptable/unacceptable -- or a numeric score for more advanced TPC management programs that may allocate weights to various aspects critical to an organization's business.

Mitigate or Accept Risks Arising from Vulnerable TPCs: With access to the risk profile for a TPC, an organization must decide whether its use is acceptable or whether it needs mitigations. Mitigations can be done in a number of ways. The most straightforward is to look for an updated version of the same TPC or an alternative TPC that has an acceptable risk score. However, upgrading or changing TPCs may be difficult: the TPC in question may be providing a

19 https://safecode.org/managing-security-risks-inherent-in-the-use-of-third-party-components/

20 https://nvd.nist.gov/

unique functionality or it could have been incorporated in the legacy code already. In such circumstances, mitigations should aim to bring down the impact of risks. For example, vulnerabilities in TPCs could be mitigated by strict input validation and output sanitization by the embedding software or by reducing the privileges or access of code involving the TPC. Mitigation may also include hardening TPCs, such as by disabling unused services, changing the configuration of a TPC or removing unused parts of it. In the event that a patched version is not available for an OSS TPC, the organization using the component can submit a patch to the managing entity or "fork" the OSS code and remediate the vulnerability itself. There are pros and cons to each approach.

Monitor for Changes: Once a TPC is incorporated in a product, it needs continuous monitoring to ensure that its risk profile remains acceptable over time. Discovery of new vulnerabilities in a TPC or the TPC reaching end of life are scenarios that may tip the TPC's risk profile to unacceptable. This step should leverage public resources such as the TPC provider's website and vulnerability databases

Track Your Security Work

Any organization that does software development requires a system and process for software version control and bug tracking. There are many systems to choose from, both commercial and free/open source, some hosted by the software development organization and others hosted in the cloud. Organizations choose systems based on a variety of factors and, with appropriate configuration, any system can be used to facilitate and support secure development.

If your organization seeks to create and deliver secure software, you should consider security problems to be software bugs and track them using your normal software bug tracking system. Taking this approach, rather using a separate tracking system for security bugs, or worse, not tracking security bugs at all, helps to surface security as an issue that developers must address rather than "someone else's problem." The objective of a secure development process is to make secure software the normal product of the development team, so anything that encourages the developers to be aware of and address security problems is an asset.

To configure a bug tracking system to help manage security bugs, it's necessary to record the effects of security bugs and their impact on users. Security bug effects are categorized as:

- Spoofing (impersonating a person, process, or system)
- Tampering (modifying information without authorization)
- Repudiation (avoiding accountability for an action)
- Information disclosure (gaining access to information without authorization)
- Denial of service (interfering with the operation of a system or service)
- Elevation of privilege (gaining unauthorized control over a system or process)

The acronym for remembering these classes of effects is STRIDE, and it will become familiar to development teams that use a technique known as "threat modeling" for design analysis. In addition, you'll want to include two other effects: one for mitigations that reduce the likelihood that vulnerabilities can be exploited or make your code more robust overall, and one for requirements that are part of your secure development process (see below).

The impacts of security bugs are categorized in terms of a severity rating system, as discussed in the next section. Severity ratings depend in part on security bugs' effects, and that's a major reason for recording effects in the tracking system.

In addition to individual bugs in software, the bug tracking system should be used to record and track mitigations and proactive security measures that are part of your secure development process. For example, the need to create threat models that facilitate design analysis gets recorded as a bug that's "fixed" when the threat model is completed. And individual design issues that the threat model surfaces get recorded as bugs that will be fixed when the necessary design changes have been made. Requirements to run code analysis tools and fix the vulnerabilities they find are treated similarly. Some vendors sell products that make it easy for you to add bugs to your tracking

system that correspond to the requirements of your secure development process - as your process matures, it's worth considering such products.

A bug tracking system that records security bug effects and impact is a major asset to a development team. By querying on severity, developers or program managers can determine which are the most important bugs to fix. By querying on effect and other factors such as the technical characteristics of the bug, the security team can identify systemic problems that need to be addressed by additional tools or training. And when the time comes to release a product to customers or operations, a query for security bugs can enable a ship decision based on data rather than intuition or guesses. The next section discusses severity ratings and "bug bars" that help to determine whether or not a product is ready for release.

Have a Rating System and a Bug Bar

It is inevitable that software will have security bugs, and occasionally there may be a fair number of security bugs to contend with. It is important to have a system to prioritize the order in which discovered bugs are fixed, especially for resource-constrained organizations. To create this rating system, or "bug bar," think about the severity of different vulnerabilities and recognize that not all vulnerabilities are created equal. Which vulnerabilities are most easily discovered and exploited by an adversary? Which vulnerabilities will have the greatest impact on either the function of the product or the security of the data it contains? Are the potential impact and exploitability high enough to delay shipping the product until a fix is made? Keep in mind that ease of discovery and exploitation are commonly underestimated.

Use this information to set a minimum level of security acceptability for shipping and to create a bug bar that establishes severity levels for vulnerabilities. While many organizations in the industry use the CVSS severity rating system[21] a simpler rating system may be better for a smaller or less well-resourced development organization. The end goal is to develop a systematic way of triaging vulnerabilities that facilitates risk management and helps ensure the most severe bugs are fixed first. Microsoft has released a sample of its bug bar for a wide range of desktop and server products at SDL Security Bug Bar (Sample) - Security Documentation | Microsoft Docs[22]. While few organizations will need such an elaborate system, the Microsoft bug bar does provide a helpful example of the considerations that can go into creating a bug bar.

Use of a bug bar for risk management may involve requiring management sign-off to ship a product or release a service if a security bug above a certain level has not been fixed: the higher the level of severity, the higher the level of manager that has to accept the risk. Using this sort of sign-off process has proven to be a great way to get management - and developer - support for addressing security vulnerabilities.

21 National Vulnerability Database: https://nvd.nist.gov/vuln-metrics/cvss

22 https://docs.microsoft.com/en-us/security/sdl/security-bug-bar-sample

Recommended Practices: Development Group 2 / CIS Implementation Groups 2-3

The following practices are largely designed for organizations who have a solid understanding of their risk profile and their current security practices and are looking to add software security activities into their development work. Practices discussed in this section focus on taking advantage of the security support already built into most product development tools and build environments, getting the development team on board and ready to help in the security effort, and introducing security at a high level into product design.

Caution is warranted in the interpretation of this section that places the "easy" technology practices right next to the two main non-technical practices discussed in this paper: building a security supportive culture and training product teams. Even the most experienced secure development program leaders often call these out as two of the most challenging aspects to building a successful software security program. There is no box to simply check and move on from as effective culture and training initiatives require ongoing, dedicated efforts for success. Despite this, they are placed here because it is also widely agreed that organizations that do not address these organizational needs are destined to fail at their attempts to execute more advanced secure development practices.

Do the Easy Stuff

Even once the potential risks are understood, starting or expanding a secure development program may seem overwhelming given the complexity and diversity of software development techniques and environments and the many layers of software security practices to apply. Fortunately, years of work by the security community and software industry have identified some practices that nearly everyone can benefit from, and automated many of them so that nearly anyone can take advantage of this work.

Much of this effort is rooted in industry work to identify the most common vulnerabilities that occur during the software development process, such as the OWASP Top Ten[23] and CWE Top 25 Most Dangerous Software Weaknesses.[24] While not all organizations will experience all of these vulnerabilities, they are frequently seen across a broad spectrum of software products. Due to their common occurrence, some mitigation options may already be built into the tools and build environment that developers are already using. For example, many tools support compile and build time mitigations such as ASLR and DEP and automatic banning of unsafe C/C++ APIs. Adopting these "free" secure development options will have little impact on developers and the product development process and thus is a good place to start in the effort to integrate security practices into the development process.

Motivate the Organization

As organizations mature and software development operations grow in both size and complexity, it becomes increasingly important to consider the business processes and culture that support a strong secure development program. In order to make software security work at scale, organizations need a holistic software security program that both prescribes strong technical practices and promotes a security-supportive culture.

23 https://owasp.org/www-project-top-ten/

24 https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

It is not enough to have security be the responsibility of the security leader or team. Rather, successful organizations foster a sense of collective security responsibility across all that touch product development, including the executive team. Software developers must see security as both a personal and organizational responsibility.

While culture is often perceived as a rather intangible aspect of business planning, many successful organizations do not treat a security-supportive culture as a happy accident, but rather as "the result of deliberate action aimed at instituting sustainable organizational change."[25] Some of these actions include fostering visible participation and support from the executive and technical leadership teams and creating security champions[26] programs that empower select members of development teams to be consistently involved in day-to-day security activities. Initiatives aimed at promoting a security-supportive culture are closely integrated with investments in security awareness and training.

Train the Developers

Once developers are motivated, clearly articulate security expectations and ensure they have skills and knowledge needed to meet them. Do not assume that developers have been trained in software security as secure development is not yet a focus in the majority of software engineering programs. Secure development training works best when targeted to specific roles of developers in your product environment so developers can directly apply what they learn to their daily work. In this way, training becomes not just an abstract corporate requirement, but rather a means for continued professional development and career advancement.

In addition to providing detailed technical training for developers, the entire organization should be made aware of the importance of security. If individuals do not understand why these practices are critical to the business and their role within it, they are less likely to support their implementation and ongoing execution.

Some organizations will bring in outside trainers to assist with skills development, while others will use in-house expertise, like resident Security Champions, to provide training and support. Many employ a hybrid of approaches to ensure adequate security expertise. Smaller organizations can also take advantage of community resources like Security Engineering Training by SAFECode[27] or a new secure software development fundamentals course from the Open Source Security Foundation (OpenSSF) that is available on the EdX learning platform.[28]

Use a Secure Design

Ideally, all products should be designed to protect themselves and their users' data. Designed-in bad security is expensive to fix and nearly impossible to fix quickly. However, following every established principle of secure design is not easy, even for highly skilled developers, and not something all organizations will be able to accomplish. That said, all organizations can and should begin to consider security as they design their software and take advantage of more accessible opportunities to improve overall product security.

While threat modeling is the best design analysis technique, it may prove to be challenging for an early maturity software security program, so we'll discuss that later as a Maturity Level 3 activity. A practical alternative will sometimes be to copy a secure design or architecture and make the minimum changes necessary to make it meet the requirements and functions of the target system.

In the graphic below, we list broadly agreed upon principles of secure design. While not all of these are accessible at early maturity levels, the two design principles everyone can and should follow are 1) enforce complete mediation

25 *The Six Pillars of DevSecOps: Collective Responsibility – Key Considerations for Developing a Security-Supportive Culture:* https://cloudsecurityalliance.org/artifacts/devsecops-collective-responsibility

26 https://safecode.org/category/security-champions/

27 https://safecode.org/training/

28 https://www.edx.org/professional-certificate/linuxfoundationx-secure-software-development-fundamentals (Courses are free; additional fee for professional certificate)

and, 2) use least privilege. Simply put, complete mediation requires that every operation on information by a user or process must be authorized. Complete mediation also requires that software "never trust input." That is, information supplied to a program, whether from a human user, another program, or a network interface, must be validated to ensure that it is not only requesting access that it's entitled to but also properly structured so that it cannot fool or evade security controls. Programs that validate input should always do so by looking for input that is valid or correct and accepting only that rather than attempting to detect every possible way that input can be incorrect or malicious.

Least privilege complements complete mediation by requiring that users or processes be granted only the rights required to do their job. Least privilege becomes especially important if there's an error in the software or by a user - it limits the scope of damage that an attacker can cause.

In practical terms, the design team might proceed by drawing a diagram that depicts classes of users and the classes of files that users operate on, and then ensuring that the software includes mechanisms that check operations against the users' rights. See the next section on platform security features for tips on resources that help with user identification and authorization.

Principles of Secure Design

The principles of secure system design were first articulated in a 1974 paper by Jerome Saltzer and Michael Schroeder (The Protection of Information in Computer Systems). The principles from that paper that have proven most important to the designers of modern systems are:

- Economy of mechanism: keep the design of the system as simple and small as possible.
- Fail-safe defaults: base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).
- Complete mediation: every access to every object must be checked for authorization.
- Least privilege: every program and every user of the system should operate using the least set of privileges necessary to complete the job.
- Least common mechanism: minimize the amount of mechanism common to more than one user and depended on by all users.
- Psychological acceptability: it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.
- Compromise recording: it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

In the years since Saltzer and Schroeder published their paper, experience has demonstrated that some additional principles are important to the security of software systems. Of these, the most important are:

- Defense in depth: design the system so that it can resist attack even if a single security vulnerability is discovered or a single security feature is bypassed. Defense in depth may involve including multiple levels of security mechanisms or designing a system so that it crashes rather than allowing an attacker to gain complete control.
- Fail securely: a counterpoint to defense in depth is that a system should be designed to remain secure even if it encounters an error or crashes.
- Design for updating: no system is likely to remain free from security vulnerabilities forever, so developers should plan for the safe and reliable installation of security updates.

—Excerpted from SAFECode's Fundamental Practices for Secure Software Development

—Based on the work of Jerome H. Saltzer, and Michael D. Schroeder. The Protection of Information in Computer Systems. (invited tutorial paper). Proceedings of the IEEE 63, 9 (September 1975) pages 1278-1308

Use Platform Security Features

The design of a secure product or service will almost certainly develop a requirement for the inclusion of security features that help to protect information, control users' actions, and ensure accountability. A development team that is building an application or service will almost certainly host that application on an existing operating system or platform that includes built-in security features. The platform development team is likely to have vastly more security resources than an organization that is the intended target of this document. Thus it makes sense to take advantage of the work of the platform developers rather than "roll your own."

Particular features that platforms provide include:

- Encryption: when you need to protect information that is being communicated over a network or stored in a location potentially accessible to adversaries, the best practice is to encrypt the information so that unauthorized parties are prevented from reading or modifying it. Creating secure encryption algorithms and code is extraordinarily difficult and it's easy to make catastrophic mistakes. Fortunately, modern operating systems implement standardized encryption algorithms and protocols that undergo extensive review. "Rolling your own" encryption algorithms or code is literally a "worst practice."
- Identification, authentication, and authorization: if an application needs to distinguish among users, it will need mechanisms to deal with the identities of individual users, to authenticate that claimed identities are correct, and to ensure that identified users only have the intended rights or privileges. Modern operating systems provide effective mechanisms for identification, authentication, and authorization and make those mechanisms available to applications. Using platform features in these areas will reduce developers' workload and minimize the likelihood of design or implementation errors - as with encryption, the platform developers have more resources than any application development team is likely to have.
- Auditing and logging: applications that deal with sensitive information may be required to keep track of "who did what" in the context of the application and the information it manages. Operating systems provide mechanisms to create and maintain secure audit logs. And an application that is using platform identification and authentication services will have a simpler task of recording the "who" in the audit log it maintains.

The SAFECode Fundamental Practices for Secure Software Development[29] expands on the security features that the platform provides and on the reasons why using them is a best practice.

Minimize Attack Surface

Nearly all software products have exposed interfaces, such as open ports, unprotected files or unauthenticated services, that allow users and other software components to interact with the product and enable it to function as intended. Products also are rich with features and options, many of which are needed by only a fraction of users. These exposed interfaces and features comprise the attack surface of your product. Minimizing the set of unprotected ports, services, and files and the set of default-enabled options and features that comprise a product's attack surface reduces the impact of potential vulnerabilities. To take a simple example, only allowing access to authenticated users who have been authorized is a great way to protect your software and system. If you can build your system so that the only people who can attack it are people who are known, you've gone a long way toward protecting the system.

The primary rule of attack surface reduction is simple: don't expose something if you don't have to. If an open port or service is not critical to the product's intended function, close it. If it has to be open, see if there's a way to open it only to authenticated users. Similarly, if your product has to create or operate on files, create them so that they are only accessible to authorized users or services - and especially so that only authorized users or services can modify them. Never create files that are writable by anyone on the system or anyone on your network.

29 https://safecodedev.wpengine.com/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf

When you think about enabling a feature by default, ask yourself if most users will actually use it. A good rule of thumb is that if a feature is needed by fewer than 80% of users you should let users enable it when they need it - those who don't won't be at risk of attack by someone who exploited a vulnerability in a feature they weren't using.

There are scanning tools, like configuration scanners, that can help you identify what attack surface you have exposed. You can then reduce your exposure using either system management configuration settings or code that manipulates permissions.

Recommended Practices: Development Group 3 / CIS Implementation Group 3

This final section of recommendations is targeted at organizations that have a solid understanding of their risk profile, are performing some security activities within the development lifecycle and have taken steps to engage and support their software development teams in their software security efforts. These recommendations are targeted at integrating secure development practices across the product development lifecycle to build a more process-based and systematic approach to addressing software security. They require a more advanced understanding of secure development practices and a higher level of expertise within software development teams.

Avoid Code Vulnerabilities

Vulnerabilities from code-level errors are a major source of security issues. Organizations cannot eliminate all of these errors, but they can significantly reduce their occurrence and their impact. Code-level security measures encompass the selection of languages and frameworks, use of security and testing tools, and application of secure coding techniques. They are most successful when supported by targeted training and investment in a security-aware development culture. Detailing all of these options is beyond the scope of this paper, but by outlining the key principles that should drive code-level security efforts, we hope to help readers understand and navigate the landscape of security options available to them.

Principles to Drive Code-level Security Efforts

1. **Don't get overwhelmed.** Ask for advice on secure coding and you are bound to uncover lengthy guidance documents and countless tooling options. Don't be overwhelmed by the sheer volume of options available. First, keep in mind that your technology and environment will narrow the scope: for instance there is no risk of buffer overruns in some modern programming languages and languages that emit managed code such as Java and C#. Second, remember that doing a few things very well is likely more effective than doing everything poorly. Prioritize your efforts when just starting out so that your investments align with both your capabilities and resources and then build from there.
2. **Use root cause analysis to focus your effort.** If you've been performing root cause analysis and tracking your security work, it will help you prioritize your efforts. Focus your training initiatives on methods that will help solve the problems you are seeing, or you know that other organizations like yours are seeing. Select tools and/or methods that address the types of vulnerabilities you find most often. For instance, if an organization

is seeing lots of buffer overruns even when using safe libraries, perhaps static analysis or fuzz testing (or both) should be added to the secure development process.

3. **Integrate security into development.** Security should be part of the development process, not an afterthought. If you wait for a security team to run security tools and tests after the product is built, the developers won't have time to fix the bugs because of pressure to ship. With more continuous approaches to development, such as Agile and DevOps, security will surely be pushed aside if the development process has to stop work repeatedly for security tasks. Providing developers with desktop tools so they can address security in the process of their normal work is ideal. Security testing should also be integrated with pre-deployment tests.

SAFECode and the Cloud Security Alliance (CSA) have partnered on a long-term project on integrating security into a DevOps environment, The Six Pillars of DevSecOps. Released guidance includes:

1) The Six Pillars of DevSecOps: Automaton (https://cloudsecurityalliance.org/artifacts/devsecops-automation/), which focuses on a risk-based security automation approach that distributes automated security actions throughout the continuous software development deployment cycle.

2) The Six Pillars of DevSecOps: Collective Responsibility (https://cloudsecurityalliance.org/artifacts/devsecops-collective-responsibility), which discusses how to foster a sense of shared security responsibility to drive security into a DevOps environment.

For complete details on this joint CSA-SAFECode initiative, visit: https://cloudsecurityalliance.org/research/working-groups/devsecops/

4. **Select tools and enable tests cautiously.** There is a massive universe of tool options to consider and something to meet every need and budget. There are free and open source tools from industry organizations like OWASP and the Linux Foundation, and free tool scans available for those who are building OSS code and using repositories like GitHub. There are also a large number of commercial products, as well as services where the vendor runs the test and gives you results. While the diversity of tools available is a good thing, be judicious and deliberate in your selection of tools. More doesn't always equal better. Many tools take a lot of upfront "training" for your code base and environment before they return useful results. If used without this "training" or just a poor fit for your application, tools will return false positives that create a lot of unnecessary work and frustration for developers. Even in the best case, you are likely to have to select tool output that identifies "must fix" bugs based on the risks posed to your product and users and on the tool's ability to deliver a low rate of false positives.

Following are some specific code-level security measures to consider as your organization thinks through these general principles of applying code-level controls to its software security program. These measures are in common use among mature software providers and have shown effectiveness when implemented properly.

- **Train your developers to avoid dangerous coding constructs.** Create, maintain and communicate relevant coding standards and conventions that support developers in their efforts to write secure code and use built-in platform security features and capabilities. It is important that any initiative to roll out coding standards is supported by relevant training to ensure that developers have the needed knowledge and skills to be successful. For an example of a well-thought-out secure coding standard, see the SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
- **Use safe libraries.** Many code-level security vulnerabilities result from the unsafe use of libraries or functions that are built into or supplied with the programming language - or from the use of unsafe libraries. Two common examples are the potential introduction of buffer overruns as a result of misuse of the string handling functions for C and C++ and the potential introduction of cross-site scripting vulnerabilities as a result of errors encoding input to a web page. Organizations have released free libraries that prevent the use

of unsafe string handling functions and/or provide safe alternatives (see for example Strsafe.h - Win32 apps | Microsoft Docs and GitHub - riverloopsec/banned-h-embedded at develop.) and that provide safe encoding to protect websites from cross-site scripting attacks.

- **Run code analysis tools.** Verify that developers are following secure coding standards and conventions, and avoiding some classes of secure coding errors by using tools that automate the process of checking for code-level vulnerabilities. Ideally, these tools should be used at the developer desktop during the normal build cycle so that developers receive timely feedback on potential security problems. As was discussed above in the section on principles, there are numerous code security scanners (or static analysis tools) and multiple options for using them - and of course different programming languages require different tool options or versions. Use of static analysis tools is a secure development best practice, but it can also be a big investment so it's important to seek input from other users and try the tools on your code before making a commitment.
.
- **Run dynamic testing tools.** While static analysis tools look for vulnerabilities in source code, dynamic analysis tools attempt to find vulnerabilities in a software product or component. Fuzz testing, which is a technique that provides well-structured but random input to a program, has proven especially effective at finding vulnerabilities in programs written in C and C++, and web application security testing tools are widely used to check web page security. Because testing targets a complete product or component and can take time, you'll want to integrate dynamic testing into your development process in the same way as you do other product-level testing. As with code analysis tools, it's important to evaluate alternative tools with your organization's particular technology. SAFECode has released a comprehensive series of blogs on fuzzing at Focus on Fuzzing: Fuzzing Within the SDLC - SAFECode.
- **Use code-level penetration testing.** Penetration testing is a form of manual testing that seeks to test software in much the same way a hacker might. It is better suited to finding business logic vulnerabilities than automated testing and therefore provides another layer of assurance that your code is secure. However, penetration testing requires specialized skills and can be time-consuming and expensive. While larger, more mature organizations may have these skills available in-house, many organizations outsource penetration testing to consultants. Given its expense both in terms of time and cost, manual testing penetration tends to be reserved for the most critical or security-sensitive parts of the software.
  Penetration testing may also be used to confirm the overall security quality of your software, but it's not a substitute for a secure development process: if a penetration test finds a small number of problems, that's a sign that the secure development process has been followed. If it finds a large number of problems, that's a sign that your team needs to follow the process "for real" or that you need to update your process.
- **Have a bug bounty program.** Bug bounties can be thought of as broad community-based penetration tests. A development organization makes its software available to the internet community, or to a more select set of testers, and pays a fee (a bounty) to anyone who finds a vulnerability. Bug bounties can be an effective way to find product vulnerabilities, but they are best viewed as a complement to a secure development process. Without such a process it's easy to get into the trap of paying bounty testers for finding similar bugs over and over again. Better to have a process that identifies classes of vulnerabilities and then update your secure development process to eliminate them en masse.

Threat Model

Design-level vulnerabilities are less common than code-level vulnerabilities, but they can be very serious when they happen and some are much harder to fix quickly. Threat modeling enables you to identify and address these types of design flaws before code is created. With threat modeling, you describe the design and then think about what can go wrong in a structured way. While threat modeling can take a fair amount of expertise, it is broadly considered a high-value activity and probably the best security design analysis technique. It not only allows you to map out the system

and understand its weaknesses but can also provide valuable context to downstream activities such as security testing.

Several books have been written about threat modeling, so providing a tutorial on the process is beyond the scope of this paper. In brief, the development team creates a description of a product or component, usually in the form of a data flow diagram (DFD). The team identifies the "trust boundaries" on the DFD where an adversary could provide hostile input, then reviews the DFD elements that need protection (process and data inside the trust boundary) and seeks to identify potential vulnerabilities. Vulnerabilities are characterized according to the STRIDE taxonomy introduced in the section on security bug tracking. While early threat modeling processes told developers to "think about vulnerabilities," modern processes provide guidance as to what specific kinds of vulnerabilities might be associated with specific DFD elements - thus making threat modeling much more approachable by developers who are not security experts.

There is no one right way to generate a threat model and there are a number of methodologies that provide a repeatable process capable of identifying potential threats. SAFECode's paper, "*Tactical Threat Modeling,*[30]" provides practical guidance for those looking to get started with threat modeling.

Ensure That Released Software Has Followed the Process

No matter what specific requirements you include in your secure development process, you'll want to be sure that they've actually been completed before your software is released. As was discussed above, the simple way to gain that assurance is to use the bug tracking system. If you've made all of your secure development requirements work items in your bug tracking system, and you've filed individual bugs for all the problems that your tools surface, then running a simple query will tell you whether your team is done, and if not, what's left to do.

Teams that have mature secure development processes or complex software products (or both) will often run verification tools that can quickly confirm that basic secure development requirements have been met. For example, the binskim tool enables you to confirm that you've selected the compile and build options that mitigate exploitation of code vulnerabilities. And you can (and should) configure those tools so that they report any errors they find in your bug tracking system to make it easy to confirm that you've actually met all the secure development requirements.

If your development team has done everything in your secure development process and recorded all its work in the bug tracking system, then a query won't show any unfixed bugs and your product is ready to release. But what happens if there are a few unfixed bugs - remaining vulnerabilities (real or potential) or a security requirement that's been overlooked? That's where your severity rating system comes in. It helps you decide whether the risk posed by the vulnerability or oversight is acceptable or a "ship-stopper." While most aspects of the secure development process are executed by your software development team, reviewing such oversights is one key place where your security team comes in. The security team works with product team management to explain the impact of the issue and decide whether the risk is acceptable or not. Of course, any such exception to your secure development process

30 https://safecode.org/wp-content/uploads/2017/05/SAFECode_TM_Whitepaper.pdf

requirements remains as an unfixed bug in the tracking system and should be a high priority for the next release of the software.

Once your team has applied its secure development process and released your software, it's a great idea to consider sharing the process requirements with customers. That's an easy low-cost way to gain customers' confidence in your commitment to security. Of course, it's also an implied commitment to keep applying and improving your process - but that's a best practice too.

Buyer's Perspective: Understanding and Managing the Risk of Acquired Software

Including software security in the CIS Control asks us not only to consider how to build more secure software, but also how to help those who run broader cybersecurity programs better assess and manage their risk from in-house and acquired software.

As this paper goes through a prioritized approach to building a software security process, it should become clear that there is no one single activity or test that can guarantee security. Rather, secure software is the result of the successful execution of a layered assurance process -- and this process will vary depending on things like the developer's size and maturity, as well as the technology, platform, function and focus of the software. The security development lifecycle of a large cloud infrastructure provider will look different than the process used by a small open source developer.

While this makes sense and provides a path forward for developers, it creates an obvious challenge for customers who seek to understand and manage the security of the software they acquire and use. Since there is no consensus approach for assessing the secure development process of a vendor, customers have largely been left on their own to decipher what a "good" process should look like, even when that process varies from organization to organization.

There are a three key principles that should guide the vendor assessment process:

1. Every vendor should have a clear, visible vulnerability response process. The absence of a response process or attempts to hide it behind a wall of nondisclosure agreements is an indication of a vendor that is not committed to product security and is unlikely to be prepared to deal with security problems when they occur. Such vendors should be avoided unless the customer is very certain that no security problem in the product could possibly cause harm.
2. Software developers should be committed to a process-based approach to software security. That process may be limited to just having a commitment and a plan to deal with discovered vulnerabilities, or it might span a full spectrum of activities that cover the entire software development lifecycle. Generally speaking, customers should demand a more comprehensive process for higher value and higher sensitivity applications.
3. Software providers should be transparent about their process. Most mature software providers publish or provide documentation related to their secure development lifecycle. But even smaller organizations should be willing to answer basic questions about how they handle vulnerabilities that arise. For some more critical types of software, vendors may be willing to provide specifics about the tools and techniques they use, the security bugs they test for (down to the level of individual tool-reported errors) and their bug bar for

identifying “must fix” security bugs. Some may even be able to offer third-party verification that they adhere to their own processes.

This paper should give customers a sense of what to look for in a vendor’s documentation that can help them better understand the activities a vendor is implementing and their level of maturity in software security. And while we hope this is helpful, we realize this approach is still lacking.

Fortunately, we are starting to see parts of the industry coalesce around the importance of taking a process-based approach to evaluating software security. Last year, NIST published its Secure Software Development Framework (SSDF)[31], which brought together what we have learned about software security over the past two decades and created a framework for communicating about software security activities. Buyers can use this framework to build their security requirements and understand whether a software provider’s secure development process follows best practices. Other efforts in progress, like the development of ISO 27034 and the evolution of IEC 62443, are also working toward supporting this goal.

Summary

Software security has evolved from a relatively narrow activity limited to a small group of specialists or software engineers to a core focus for organizations that have come to recognize the need to manage the security risk from both in-house developed and acquired software in order to protect against some of the most prevalent cybersecurity attacks. In recognition of this shift, CIS took the important step of adding software security to the CIS Controls.

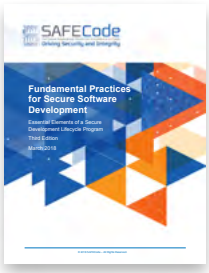
Fortunately, those looking to introduce or expand the use of software security practices have a long history of work in the discipline to help inform their efforts. While much of this guidance is easily obtained, existing software security frameworks often consist of a large catalog of practices written with large software development organizations as their primary audience. This paper does not pretend to replace that guidance, but rather aims to help organizations more easily navigate and apply it based on their resources and requirements.

While these recommendations are prescriptive in nature, it must be emphasized that there is no one-size fits all checklist for software security. Rather, root cause analysis should be a primary driver of secure development efforts to ensure that organizations direct their resources at the issues that most commonly impact their own software. Further, software security must be viewed as a holistic process that will evolve alongside the organization. We hope that the tiered approach provided here can help readers not only identify their needs today, but also map a path of continuous improvement to meet the demands of tomorrow.

31 https://csrc.nist.gov/News/2020/mitigating-risk-of-software-vulns-ssdf

For Further Reading

[CIS Controls]

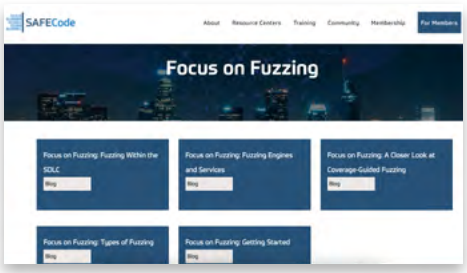


[Fundamental Practices for Secure Software Development]

[Managing Security Risks Inherent in the Use of Third Party Components]

[Software Security Takes a Champion]

[Tactical Threat Modeling]



[Focus on Fuzzing: A Blog Series]
[SAFECode Training]

[Cloud Security Alliance-SAFECode – The Six Pillars of DevSecOps: Collective Responsibility]

[Cloud Security Alliance-SAFECode – The Six Pillars of DevSecOps: Automation]

[NIST Secure Software Development Framework]

[BSA Framework for Software Security]

[OWASP]

[FIRST]

A Note from the Authors

The authors would like to acknowledge and thank the many SAFECode members who have contributed to its published guidance. Their years of effort identifying, documenting and promoting the software practices their organizations have found to be effective in improving the software security formed the foundation for this paper. We encourage readers to visit www.safecode.org to follow their ongoing work.