Viestintävirasto
Finnish Communications Regulatory Authority

SECURE DEVELOPMENT: TOWARDS APPROVAL

TURVALLINEN TUOTEKEHITYS – KOHTI HYVÄKSYNTÄÄ

Viestintävirasto 003/2018 J

Viestintävirasto
Kyberturvallisuuskeskus
Puhelin: 0295 390 100 (vaihde)
PL 313 (Erik Palménin aukio 1)
00561 Helsinki

Finnish Communications Regulatory Authority
National Cyber Security Centre Finland (NCSC-FI)
Phone: +358 295 390 100 (switchboard)
P.O. Box 313 (Erik Palménin aukio 1)
FI-00561 Helsinki

www.ncsc.fi
www.ficora.fi

TABLE OF CONTENTS

ESIPUHE

Kyberturvallisuudella on merkittävä rooli sekä yhteiskunnalle kriittisten järjestelmien ja toimintojen turvaamisessa että kansalaisten arjessa. Tehokas tapa parantaa kyberturvallisuutta on puuttua mahdollisiin ongelmiin jo ohjelmistotuotteiden ja -palveluiden kehitysvaiheen aikana. Turvallinen tuotekehitys edistää toimintavarmuutta ja ennaltaehkäisee tietomurtoja ja -vuotoja.

Viestintäviraston Kyberturvallisuuskeskuksen kansainvälisiin tietoturvavelvoitteisiin kuuluu salaustuotteiden hyväksyntä kansainvälisen turvallisuusluokitellun tiedon suojaamiseksi Suomessa. Velvoitteen täyttämiseksi Kyberturvallisuuskeskuksen National Communications Security Authority (NCSA-FI -toiminto) on arvioinut salaustuotteiden teknisiä toteutuksia ja niiden valmistajien tuotekehityskäytäntöjä. Arvioitujen tuotteiden tietoturva on parantunut, ja työ on osoittautunut tehokkaaksi osaksi ennaltaehkäisevää kansallista kyberturvallisuustyötä ja suomalaisten salaustuotteiden myynnin edistämistä.

Kevään 2018 aikana NCSA-FI toteutti selvitystyön turvallisen tuotekehityksen ja hyväksyntään valmistautumisen tukemisesta. Selvitystyössä valmisteltiin "Turvallinen tuotekehitys - kohti hyväksyntää" -opas ja suunnitelma ennaltaehkäisevästä tuoteturvallisuustyöstä viestimiseksi. Oppaan valmistelussa hyödynnettiin sekä NCSA-FI -toiminnon kokemusta että teollisuuden asiantuntijoita.

FOREWORD

Cyber security has a significant role in protecting society, its critical systems and its citizens. An effective way to improve cyber security is to address these potential problems early during the development stage of producing software-based products and services. Secure development improves robustness, ensures continuity, and helps prevent data breaches or leakages.

Duties of the FICORA's National Cyber Security Centre (NCSC-FI) include approving cryptographic products for protecting international classified information in Finland. In order to fulfil this obligation, the National Communications Security Authority (NCSA-FI), operating as part of the NCSC-FI, has carried out assessments of cryptographic products and their development processes. This has helped to improve the security of these products. Furthermore, this work has proven to be an effective and proactive contribution to national cyber security, while also promoting sales and exports of Finnish security products.

In the spring of 2018, NCSA-FI began exploring how better to support vendors in secure development and methods for preparing their products for assessment and accreditation. As part of this work, this guidebook called Secure Development: Towards Approval and plans for what information to include with this guidance were developed. Both industry experts and NCSA-FI's own experts participated in developing the guide.

YHTEENVETO

Tämän oppaan tarkoitus on auttaa valmistajia tekemään laadukkaita ja turvallisia tuotteita. Opas on suunnattu Kyberturvallisuuskeskuksen NCSA-FI salaustuotehyväksyntää hakeville ja muille tietoturvasta kilpailuetua tavoitteleville suomalaisille valmistajille. Opas tehostaa hyväksyntään valmistautumista, antaa neuvoja turvallisesta tuotekehityksestä ja tukee tietoturvatuotteiden kehitystä viranomaiskäyttöön ja vientiin.

Tuotteen arkkitehtuurissa ja suunnittelussa turvallisuutta lisäävät hyväksi todetut suunnitteluperiaatteet: hyökkäyspinta-alan minimointi, turvalliset oletusarvot, ulkopuolisten syötteiden tarkistus, oikeuksien minimointi, syvyyssuuntainen puolustus, turvalliset virhetilat, epäluottamus ulkopuolisiin palveluihin ja turvamekanismien yksityiskohtien salailuun pohjautuvien oletusten välttäminen.

Tämä opas on yhteenveto turvallisen tuotekehityksen vaiheissa huomioitavista asioista.

Tietoturva on tietoturvaominaisuuksia, esimerkiksi salaustoimintoja, mutta myös ohjelmiston laatutekijä. On siis tärkeää kiinnittää huomio myös salausta laajemmin tuotteen eri toimintoihin, ja ymmärtää että myös ne kuuluvat hyväksynnän piiriin. Ominaisuuksien lisäksi tietoturvallinen tuotekehitys kattaa myös kehityksen ja ylläpidon aikaiset toimet, kuten tilaturvallisuuden, käytettyjen järjestelmien turvallisuuden sekä tuotekehityshenkilöstön koulutuksen. Itsearviointi Katakri-auditointityökalun ohjeistuksen avulla antaa hyvän pohjan kehitysympäristön ja -organisaation valmistelemiseksi hyväksyntään.

Uhkamallinnus on tuotteen suunnittelu- ja päivitysvaiheiden tärkeimpiä työkaluja. Uhkamallissa kuvataan tuotteen käyttötapaukset, ympäristö uhkien näkökulmasta, järjestelmän tuottamat arvokkaat tiedot ja palvelut, ja kuinka tuote vastaa näihin kohdistuviin uhkiin. Tärkeintä on, että uhkamalliin liittyvät asiat käydään läpi osana tuotekehitystä, eikä se millä metodilla uhka-arvio tehdään. Uhkamalli tukee myös tarkastustoimien tehokasta rajaamista ja kohdistamista.

Dokumentoidut ja omaksutut suunnitteluperiaatteet helpottavat sekä turvallista toteutusta että toteutuksen turvallisuuden arviointia.

Tuotteen toteutusvaiheessa on tärkeää varmistaa turvallista tuotekehitystä tukevat työkaluvalinnat, toteuttajien tietoturvaosaaminen sekä valittujen kolmansien osapuolten komponenttien ja alustaratkaisujen turvallisuus. Monet tietoturvaongelmat syntyvät ohjelmointivaiheessa. Turvalliset tekniikat sekä niiden puutteet riippuvat käytetyistä alustoista, komponenteista, ohjelmointikielistä ja työkaluista. Kaikkiin näihin tulee perehtyä. Materiaaleja tietoturvalliseen ohjelmointiin, riippuvuuksien turvallisuuden arviointiin ja alustojen koventamiseen on yleensä hyvin saatavilla. Kolmansien osapuolten komponenteista löytyy ja julkaistaan haavoittuvuuksia säännöllisesti, joten tietoturvan tason säilyttäminen vaatii tuotteen päivittämistä. Päivitykset puolestaan voivat vaatia uudelleenhyväksyntää, jolloin kehitys- ja päivitysprosessin kypsyyden merkitys korostuu arviointitoiminnassa.

Ohjelmiston laadunvarmistukseen, siis myös tietoturvaan, kuuluu kattava testaus. Testausta pitää suorittaa todellista käyttötilannetta vastaavassa ympäristössä ennen kuin tuote tulee hyväksyntään. Testauksessa ja laadunvarmennuksessa tulee pyrkiä kohti toistettavia ja automaattisia menetelmiä, koska niillä saavutetaan suurempi testikat tavuus, ja järjestelmään tehtäviä muutoksia pysytään näin testaamaan tehokkaasti ja luotettavasti. Tuotteen ja sen osakokonaisuuksien helppo testattavuus nopeuttaa myös sen hyväksyntää. Myös katselmoinnit ovat tärkeä osa laadunvarmennusta, ja tuote pitäisi katselmoida myös tietoturvaperspektiivistä. Toteutetut testaukset, itsearvionnit ja mahdolliset kolmansien osapuolen suorittamat tarkastukset tukevat hyväksyntään valmistautumista.

Tämä opas on yhteenveto turvallisen tuotekehityksen vaiheissa huomioitavista asioista. Tämän lisäksi on välttämätöntä perehtyä oman erikoisalan ja valittujen työkalujen ja alustojen tietoturvan erityispiirteisiin. Jos voidaan todeta valmistajan tuotekehitystyökalujen ja -menetelmien, testauksen, tilojen ja kehittäjien osaamistason olevan kunnossa, asiakkaiden luottamus tuotteeseen parantuu ja hyväksyntä nopeutuu.

EXECUTIVE SUMMARY

The purpose of this guide is to help vendors to create high-quality, secure products. It is aimed at organisations applying for approval of cryptographic products from NCSA-FI and at other Finnish vendors seeking to gain a competitive advantage from information security. This guide helps readers to better prepare for the assessment, provides insight for anyone interested in secure development, and supports the development of products for govemental use and export.

This guide provides a summary of the phases of secure product development.

Security is made up of features such as encryption, but security is also a quality attribute of the system. Therefore, you should pay attention to all features, not just to the cryptographic functions. A security assessment will look at your product as a whole. Beyond features, secure product development encompasses the development process itself, including the maintenance phase. The development facilities and tools must be secured, and the staff must be trained. Doing self-assessment using the Katakri auditing tool gives a good foundation for preparing both the development environment and the organisation for passing the approval process.

Threat modelling is one of the most important tools for designing and maintaining your product. The threat model describes the use cases of the product, the threat environment,

the valuable information protected by the system, and the services offered by the system. Threat models help you to assess how the product counters threats. There is no single correct way to do threat modelling; the important thing is to do it and to leverage the results to support better secure product development. The threat model also supports the effective specification and targeting of assessment activities.

Product security is enhanced by using established architecture and design principles: minimal attack surface, safe defaults, input sanitation, minimal privileges, defence in depth, failing safely, not trusting external services, and avoiding security by obscurity. Adopting and documenting design principles facilitates both secure implementation and security assessment.

When implementing a product, the tools you use should support secure development, developers should be security-aware, and third-party components and platforms should be secure. Many security issues arise during the programming phase. Secure programming techniques and security pitfalls are specific to the platforms, components, programming languages, and tools used. You must be familiar with all of these. Material for secure programming, dependency security assessment, and platform hardening are generally well known and easily available. Vulnerabilities in third-party components are found and announced regularly, so maintaining the security of the product requires constant updates. Updates may lead to re-approval, which places even more emphasis on maturity in the development and maintenance processes as an enabler for an efficient approval process.

Comprehensive testing is part of software quality assurance, including security assurance. Testing needs to be carried out before the product is submitted for approval, in an environment that matches the real-world situation. Testing and quality assurance should aim for reproducible, automated methods, as they provide greater test coverage and enable efficient and reliable testing of the changes to the system. Products and components that have good testability will speed up approval. Code reviews are an important part of quality assurance, and the product should also be reviewed from the security perspective. Carrying out various tests and self-assessments, and receiving third-party assessments improve the approval process.

This guide provides a summary of the phases of secure product development. On top of that, you need to familiarize yourself with the specifics of your domain and the tools and platforms you are using. After all, when a vendor's development tools and methods, testing, facilities and developer skills are in good shape, customers' confidence in the product will improve – and, again, the approval process will be faster.

SECURITY MATTERS

This guide is intended to help you to create high-quality, secure systems. It gives you an outline of how to design, implement and test secure systems. It is not only developers that need to understand secure development; support from R&D managers, product managers, or other people involved with product development is also essential. In addition to secure development guidance, we will also give you practical tips if you are applying, or planning to apply, for approval of cryptographic products from NCSA-FI.

This guide is intended to help you to create high-quality, secure systems

SECURITY IS A COMPETITIVE ADVANTAGE FOR YOUR COMPANY:

- Heightened awareness about cyber security and privacy issues has led customers to demand security and high quality from these products and services, even when they do not have the means to verify it.
- If you have been proactive, you are well positioned for security-related tender requirements. You will be ready when the sales manager says that proof of secure development practices is required for the "customer meeting next week".
- If you have done your part in securing the system, your customers are less likely to be compromised. Your customers are less likely to hold you responsible if you have demonstrated due diligence.
- Responding to a security incident is daunting and expensive.

Secure development reduces your risks, is good for continuity of your business, and improves the quality of your work in general. Security as a business practice has an overall positive impact.

Often, especially in the past, the security of many software-based products has been low. We have often witnessed a situation where a vendor wakes up to security considerations late in the game, for example in the bidding phase. Considering security attributes of products late in the development cycle is painful for everyone: vendors, auditors and buyers. It appears that security is not always communicated as a requirement from the management, and therefore is not always as proactively considered as one might think.

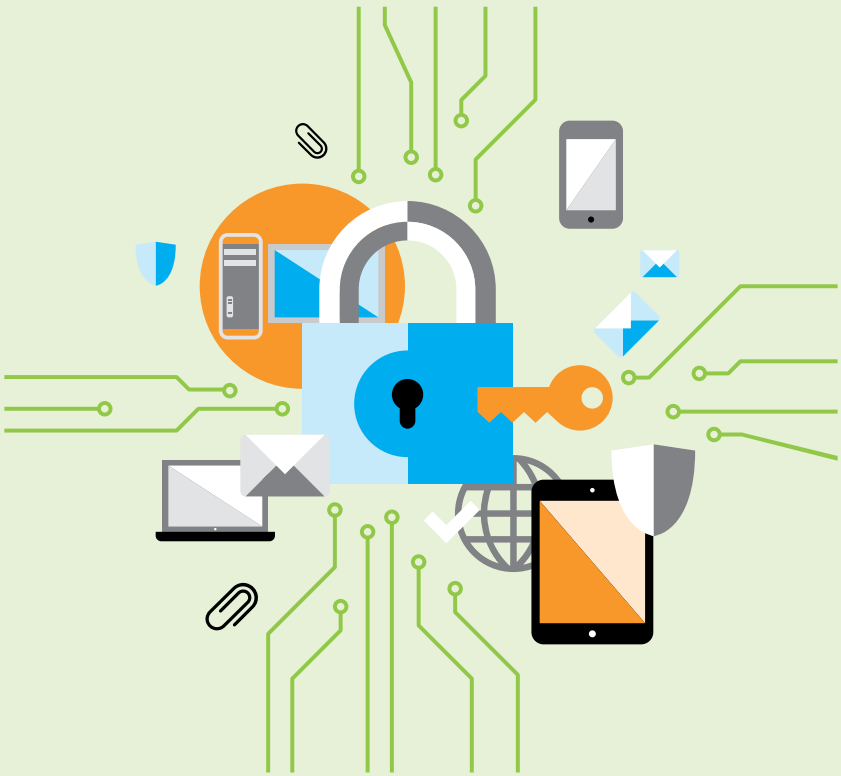
HINT:

Preparing for security audits has a positive impact on your product:

- If management is expecting you to pass the audit, in exchange they should be ready to invest in security, quality and continuity, and to make them part of the company culture.
- Proactive work will make the audit easier to pass.
- The assessment itself may discover flaws to fix and new ways to improve your product.

Further reading

- *"Computer security is broken from top to bottom", The Economist, 8 April 2017*
- *"Why Cybersecurity Should Be a No. 1 Business Priority For 2017", Forbes.com, 20 March 2017*



FACILITIES AND PERSONNEL – OPERATIONAL SECURITY

Secure products are made in secure environments. It is the people who design and implement the software, and people need tools to work with and places to work in. If the facilities holding source code, build artifacts, tools or work computers are compromised, the attacker can compromise the products manufactured there. For example, a backdoor could be added to source code. We should think of all parts of the system – including personnel and facilities – when we think about security.

The tools used for development should match the security requirements of the products being developed, e.g. version control that carefully tracks all edits and who did them is a must-have. Servers should be up to date with the latest security patches, and all users should have unique accounts for logging purposes. Some popular cloud services aimed at software developers may be doing a better job in keeping their platform secure and updated than organisations that have too few resources to secure their in-house development platform.

People should know general principles of information security hygiene.

Popular secure software development life-cycle models require developer training. This should be extended to all personnel involved. People should know general principles of information security hygiene, e.g. don't click everything you receive by email, be careful when browsing and perhaps do not use work computers for leisure browsing at all, do not pick up and plug in USB sticks that you find lying around, keep your systems up to date, and understand the basics of social engineering. Developers should get additional training for secure design, threat modelling and secure programming.

People come and go, so training must be a recurring part of the onboarding process for new hires. Relying on one or two "hero" programmers is not a secure strategy for the product or for the company.

Further, laptops holding development material should be up to date with the latest security patches and have disk encryption.

Besides the facilities and personnel, you must think about the processes used in your organisation. Do they support the development of high-quality, secure products? A quote from OWASP (https://www.owasp.org/index.php/Policy_Frameworks): *"...for a secure application, the following at a minimum are required:*

- *Organizational management which champions security*
- *A written information security policy properly derived from national standards*
- *A development methodology with adequate security checkpoints and activities*
- *Secure release and configuration management processes"*

OWASP also states the following: *"Ad hoc development is too unstructured to produce secure applications. Therefore, organizations who wish to produce secure code consistently need to utilize a methodology that supports that goal. Choose carefully – small teams should never consider heavyweight methodologies that identify many different roles, while large teams must choose methodologies that will scale to their needs."*

HINT:

You could use the Katakri framework (information security audit tool for authorities) for self-assessing your product development facilities, processes, and organisational structure. It can be used by anyone as a checklist for securing an organisation and its facilities. Any external assessment is likely to follow similar paths, and if you are in line with Katakri, you are well positioned for such assessment.

Further reading

- *"Katakri 2015 - Tietoturvallisuuden auditointityökalu viranomaisille (available in Finnish and in English)"*

REQUIREMENTS AND THREAT MODELLING

Software requirements define the expected functionality of the software. They come in the form of use cases or lists of requirements. On small projects, the requirements may be quite informal and perhaps not even written down.

Coming up with the right set of requirements is hard, but it is essential for the project's success. Doing it entirely at the beginning of a project, before design and implementation, is usually impossible, as both developer organisation and customers learn more as they go. Nowadays the trend is towards iterative work done in cycles of gathering requirements → design → implementation → verification. Security requirements may be even harder to define than purely functional requirements. "The software should not crash" is a desirable goal for secure software, but usually too generic as a verifiable requirement. We will take a more detailed look at defining security requirements with the help of threat modelling.

SECURITY REQUIREMENTS

Sometimes requirements are inadequate or missing, and this leads to wasted time in design and implementation, and shortcomings in the resulting software. The same goes for security requirements. They define how the information and services of the system are protected from malicious actors and other misfortunes. For example, security requirements may determine how users are authenticated or what data in the system needs to be encrypted.

As with any other requirements, the security requirements can be functional as well as non-functional. For example, a functional requirement could state that users must first log in with a username and password. A non-functional security requirement could be that the application must validate all input received over a network and drop all invalid requests.

To come up with robust security requirements, we must envision how the product is really going to be used and in what sort of environments. Is the system going to be physically isolated in a bunker with guards? Will it be hosted in a cloud service? Does it have components running in a Web browser? What assets of value (usually data) will the system handle?

HINT:

Write down your security requirements, even the implicit ones. Written security requirements will make your product more secure and security assessments more effective.

Once we have laid out the intended usage of the product, we can then consider what could go wrong. Can the product be misused? If so, by who, how and when? Can an attacker get something they should not have or make you lose something valuable? Could someone who is not authorised access the servers to steal data or even physical disks? What if the browser component is reverse-engineered and replaced with a malicious client? What would be the impact on the system being compromised?

The ways to compromise a system are numerous, and attacks are varied. However, there are general categories and characteristics for most known attacks. For example, the OWASP Top 10 web application security risks lists the following vulnerability categories, which are mostly relevant outside the web application area as well.

1. **Injection:** The application accepts external input, but does not validate it properly. This allows an attacker to execute commands or do other misdeeds in the vulnerable application.

2. **Broken authentication:** User authentication is not implemented correctly. Passwords are not verified, passwords are leaked, or password recovery may be used to attack the system. Session handling after authentication may also be broken in a way that allows an attacker to hijack sessions.

3. **Sensitive data exposure:** Sensitive data is stored or transported in clear text or using only weak protection.

4. **XML external entities (XXE):** The application has insecure XML processing, such as misimplemented SAML for single sign-on.

5. **Broken access control:** The application allows users to perform actions outside their intended permissions.

6. **Security misconfiguration:** A system is missing security hardening or has unnecessary services running, or the platform is old and vulnerable or has insecure built-in accounts.

7. **Cross-site scripting (XSS):** Attackers can execute unwanted HTML or JavaScript.

8. **Insecure deserialisation:** Attackers can tamper with data/objects that the application deserialises and then uses for privileged actions.

9. **Using components with known vulnerabilities**: The application usescomponents, intentionally or by mistake, that are known to be vulnerable.

10. **Insufficient logging & monitoring:** The application does not hold sufficient secure logs for investigation or the application does not monitor or provide alerts for attacks.

"Misusers" come in different types – from script kiddies to profit-seeking criminals to nation-state actors – and each of them has different capabilities. Which one you must consider depends on the usage of the planned system. You should also think about what motivation a person could have to compromise the system and the valuable assets the system holds, as well as what the impact of a successful attack would be for your customers and for you.

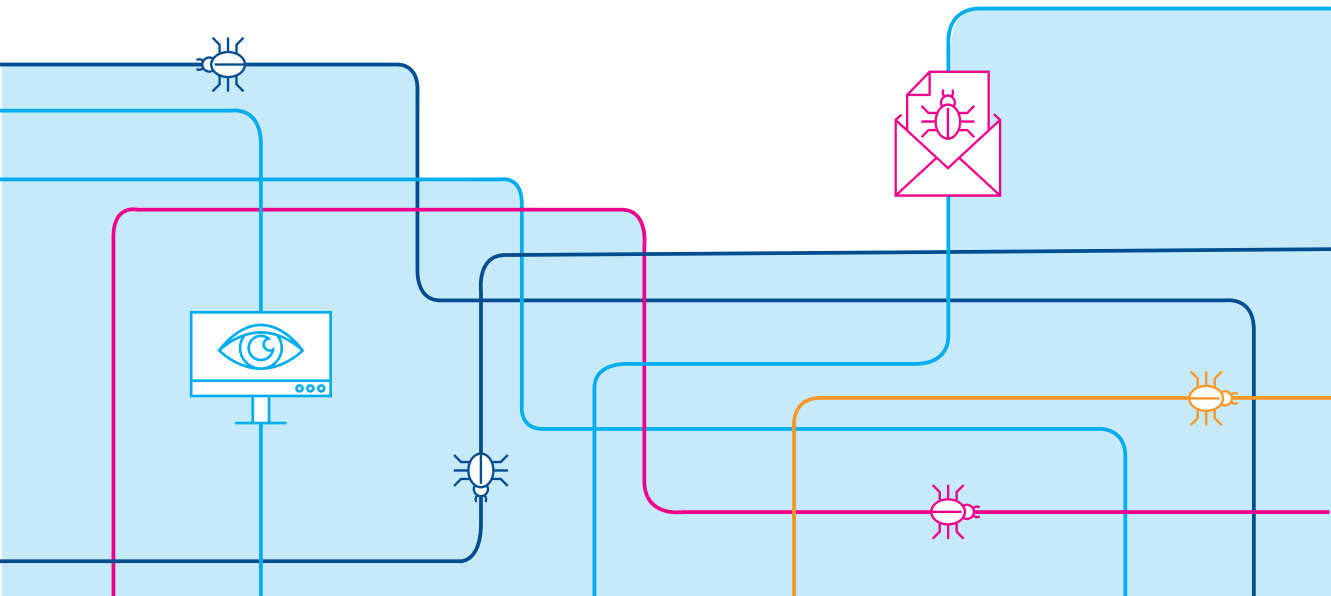
THREAT MODELLING

Threat modelling is the term used for evaluating the security of a system. When discussing the security requirements we already outlined some general principles of threat modelling. There are several methodologies for threat modelling, but it does not require special skills or learning some complex methodology.

"There are multiple approaches to threat modeling... The method used to assess risk is not nearly as important as actually performing a structured threat risk modeling. Microsoft notes that the single most important factor in their security improvement program was the corporate adoption of threat risk modeling." - Threat Risk Modeling - OWASP

It is not so much about how you do it, as long as you do it. Think like the attacker, and build security requirements into your action plan. You need a high-level architecture in your system for threat modelling, as different components have different roles and different security characteristics, and they hold information with varying levels of security. As architecture design is usually considered as part of the design, you need to iterate between design and threat modelling, which again calls for an iterative development process.

If you have trouble getting it done, consider using external consultants to facilitate it.



HINT:

If your product is going to be audited for security, then threat modelling is going to be part of the audit. Auditors may focus on a specific subset of features in your product, and they will most likely have a specific usage scenario and environment in mind. For example, you may have 20 features in your product, but only five of them may be audited. The remaining 15 features may need to be disabled in the audited usage, unless they have no overall security impact.

Auditors' threat model may not match yours. They need to focus on the threats that are relevant for their assignment. As the supplier of technology, you should understand what your customer requires and work with them to build the relevant threat model. What kind of data at which security level are they going to handle with your system? What happens to you customer if security is breached? Who are the crooks or enemies your customer will need to worry about? Who are the future users of the system?

The auditors are most likely going to use the system from their angle, emphasising the use cases and scenarios relevant to their assignment. They will observe how the system behaves and tie back to the threat model that they are building. Does the threat model cover the actual usage scenarios and features of the product?

This is good for you, if you already have a system that you are not totally confident with security-wise. By understanding the priorities of the customer who is going to assess your product, you get a to-do list for priorities for your security update. The things that are not immediately important for the customer can be pushed to future projects.

BUILT-IN SECURITY VS. ADD-ON SECURITY

Often the term built-in security is used to refer to the principle that security requirements are taken into account from the beginning of a development project, resulting in a product that has security embedded into it. The other principle is add-on security, where security is considered only after the product is implemented, and the security is provided by adding new components and features for protection.

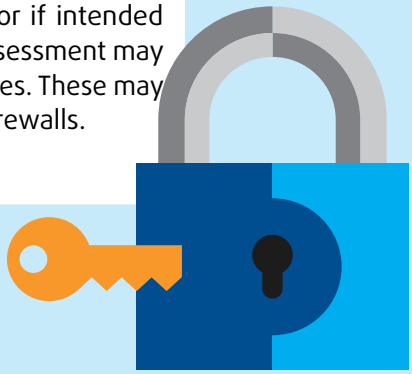
Built-in security is considered better for several reasons. Adding new components always brings integration problems, additional costs, and increased likelihood of vulnerabilities. With built-in security, this is avoided, as security is an integral part of the components. Furthermore, security is a quality attribute alongside maintainability, testability, throughput and usability. It is difficult to instil quality by adding new components.

Considering security from the early stages and training your development staff helps to bring a security mindset into teams and becomes a great asset in later phases of the project. Conducting security assessments, secure deployment and security patches are the foundations of a secure product.

Add-on security, such as firewalls, application sandboxing and intrusion detection systems, can give you further assurance. However, if your product has not been built with a strong and secure development focus, they are only stopgap measures. In that case, we recommend you begin "building security in" as soon as possible to upcoming releases.

HINT:

If your product's own security falls short or if intended use cases are especially challenging, the assessment may require you to add further protective measures. These may include measures like physical barriers or firewalls.



PRIVACY

Related to security is privacy and protection of personal data. Privacy requires security, but issues like how long you can store personal data for or if you need to inform your customers about the data you are storing about them are excluded from this guide. We just want to remind that you may not have the luxury to ignore these issues.

Further reading

- *Synopsys, June 2016: Are You Making Software Security a Requirement?*
- *OWASP: Threat Risk Modeling*

DESIGN

Once you have laid out the requirements, it is time to design the system. Hopefully, you are using an iterative approach and you revisit the requirements and design phases several times. This makes the job easier, as nothing will be perfect on the first run.

Design determines the architecture of your system needed to deliver the intended functionality, as stated in the requirements. However, the architecture also has a massive impact on the ease (or difficulty) of making the system secure. As you perform threat modelling with your architecture plan, you will most likely spot architectural changes that will make your product easier to secure.

SECURE DESIGN PRINCIPLES

There are general "rules of thumb" for secure design. The ones below are adapted from OWASP Security by Design Principles.

Minimise the attack surface

The term "attack surface" is used to refer to the portion of the system that is exposed to the outside world, physically or through a network or files. All attacks are expected to come through this surface, unless you have missed something.

You should consider ways to minimise the attack surface by eliminating non-essential interfaces. For example, does the device need a USB port, or can it be covered to make access more difficult? Can some default services of the platform OS be disabled to harden the platform? We will return to the topic of platform hardening later. Can administrator access only be allowed from a local host? Perhaps the system Web service can be made with static pages only without dynamic processing on the web server itself?

Reducing the attack surface has many benefits. There will be fewer threat scenarios to consider and less room for implementation mistakes, and testing the system is easier.

Establish secure defaults

Your customers should not have to be experts to use your system securely. Sadly, systems are usually insecure by default, and securing them is left for the administrator of the system. As an administrator is not as familiar with the system as the developer, the system should ship with secure defaults and minimal configuration. Any weakening should be a conscious decision made by the user of the system. Think about your liability. Would you like to take responsibility for shipping an insecure system by default, or would you rather let the administrator make the decision to weaken the security after a risk assessment?

Sanitise input[1]

Failure to validate input may allow an attacker to corrupt or crash the vulnerable parts of the system. Moreover, it is often possible to execute commands in the system or otherwise gain control of it. Whenever possible, validate input coming to your system from outside. Validation means that the syntax of the input (such as messages) conforms to expected rules and the input is semantically valid.

Encryption and integrity checks are often used to protect the data in transit, but unless you absolutely control the origin of the input data, even encrypted input needs to be rigorously validated.

[1] Not in the OWASP list

You should also consider indirectly exposed interfaces. Messages from the attacker could be carried deep to the core of the system. Here is an example of input propagation:

1. A web server receives a username and password (credentials).

2. It uses an authentication helper to forward the credentials to an authentication server.

3. The authentication server does a database lookup.

4. The username is recorded in the audit log.

5. The administrator studies the audit log with his/her browser.

Consider all the interfaces. Make sure that at least someone understands and documents the data flows in the system. Train your team to assume that malicious input could reach their components, no matter where they are.

Fuzzing is a great way to test input sanitation. We will cover that topic in more detail in the testing portion.

Make sure that at least someone understands and documents the data flows in the system.

Separate duties

Separation of duties takes place when one person submits a travel expenses claim, but a different person is required to review and accept it. The same logic applies to software components. A classic example would be moving responsibility for audit logs to another system that cannot be compromised along with the system that produces the logs. You should not store audit logs of database activity in the same database. If the database is compromised, you will not be able to trust the audit trails.

Separating duties into different components also allows you to do more granular tuning of the resources and privileges available for the component.

Give minimum privileges

Once you have separated duties between the components, you should minimise the access rights of these different processes or sub-systems. Each component should have the minimum set of access rights to complete its intended mission. The same applies to users: not everybody should have administrator access, and administrators should not be all-powerful either. With proper least privileges assigned, compromises are more likely to remain contained and the attacker's access limited. Or you can think of it the other way around: if a component can operate with minimal access rights, then the security of the component is less critical than in a situation where the component has admin rights.

HINT:

In the assessment, you do not want to have to confess that all your code is running with the highest possible privileges. You should be able to articulate what privileges are used and where.

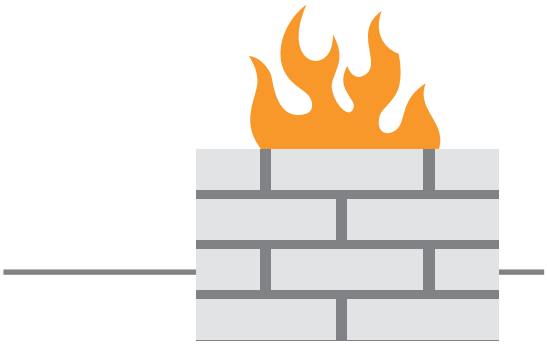
Defend in depth

Defence in depth means that you design multiple layers of defence into the system. All security controls can contain compromising errors. Also, a determined attacker will find ways around specific controls. This way, you make the system more secure, as compromising a single layer does not compromise the whole system.

For example, you cannot assume that a firewall alone will keep you safe. A firewall typically has rules to pass traffic. A malicious program may get through to your protected perimeter via email. A laptop typically connected to your network may pick up an infection on a business trip. Users browse the web and can be compromised by malicious websites, giving the attacker a bridgehead to your network. Defence in depth means that even with a firewall in place, you also harden your internal network, encrypt internal traffic and require authentication to access internal services. This way, you are not vulnerable even if the firewall is compromised.

There should be no system account that allows unlimited system access; instead, there should be different user roles with the principle of least privilege. User actions should leave audit trail that cannot be tampered with. Access to at least the more powerful roles should require two-way authentication.

Sometimes the term deep security is used to mean the same thing.



Fail securely

Be prepared for failure. Hardware breaks down; network connections fail; batteries run out; software crashes; and so on. You should design your system in a way that such a failure does not compromise system security. Sometimes this can be tricky. You have two strategies to choose from: fail open vs. fail closed. Should the system grant or deny access when it fails? If component authorising user access cannot be reached, you may always want to deny access. But if denying access would lead to drastic consequences – such as polluting the water supply of a large population – you would need to think twice about which strategy you choose. Our main message is: do not overlook what happens when a component fails.

Do not trust external services

It is likely that your system uses services from external systems. The software, facilities and personnel powering these services are not under your control. Third-party services can also be compromised, so don't give the attacker a means to move forward.

In the spirit of least privilege and defence in depth, you should not blindly trust external services. You should treat them as an external actor, validate all data from them, and fail securely if the service is not available.

Be open-minded about what constitutes an external service out of your control. Just think about a browser-based system: the user's browser is an external service running part of your code. You cannot rely on security controls implemented in code running on the user's browser. All your security controls must be enforced on the server. The same applies to any system where a component is running on a client system.

Avoid security by obscurity or secrecy

Security by obscurity means that security is dependent on keeping the design or the implementation of the product secret. Unfortunately, secrets leak and systems can be reverse-engineered to reveal their secrets. Do not rely on security by obscurity. While the data and access tokens your system handles might be confidential, the system itself should withstand scrutiny. Limit the secrets your system needs to fulfil its mission. Also, consider how those secrets can be changed when they leak. Is it easy to switch the private keys of the system? How easily can the owner of the system ask users to change passwords if they are compromised?

While keeping design and source code secret might provide an extra layer of security, you should never rely on it alone.

Keep it simple

All software and components can contain errors and vulnerabilities. Even security software and security features have vulnerabilities. Less code means fewer errors. More configuration options mean more configuration mistakes. Aim for your system to be as simple as possible, and always challenge the apparent need for extra complexity.

A simple system is easier to review and secure than a complex one. Further, avoiding unnecessary complexity affects your bottom line. A simple system is easier to understand, so developing and maintaining it will be more cost-effective.

HINT:

The less you have to audit, the less the audit will cost. Either keep your product simple, or be prepared to prove how only part of it is critical for the use cases to be approved.

Prepare to fix security issues correctly

When you receive information about a vulnerability in your system, it probably should be fixed or addressed. This is where good development and testing processes come in handy. You should feel confident about fixing bugs and issuing new releases of your system. With ad-hoc processes, untrained personnel and a lack of test automation, every change is a risk, and you may be tempted to ignore the problem.

We have often seen situations where a security bug is fixed only in a specific place in the system when many similar fixes are needed elsewhere in the codebase. The reason for this may be the fear of introducing side effects, so that only the minimum fix is made. Fixing one out of many similar problems is not going to help with security very much. Another pattern we have observed is that the vendor fixes the maintenance version of

the software, but overlooks the next release, allowing the problems to reappear with the next major update.

Sometimes, vendors opt to perform the minimum modifications to the maintenance version of the product and ship a larger "fix round" or refactoring for the next major release.

Time-critical bugs can also be discovered when your key developers are on holiday or sick. This is another reason to implement proper processes and training so that you can delegate fixing problems like this to the staff available at any given time.

You should feel confident about fixing bugs and issuing new releases of your system.

PLATFORM CHOICE

Your product is running on top of one or more different platforms. Some common ones are Linux systems for servers, Android for mobile devices, different cloud platforms for backend systems, Docker for sub-systems, Microsoft Windows and .NET, and so forth. Each platform has its own security characteristics, and you need to know the ones for the platform that you build your product on.

Platforms receive updates at different times for new features as security problems are discovered in them. You should consider this against the intended release cycle and life-cycle of your product. You should take a look at the release history of the platform(s) on your shortlist to get a feel for their security and update track record. Once you know the platform, you must decide whether you are going to update your product when the platform receives updates: all updates, major updates, just security updates, or none?

Leaving security updates unhandled is naturally problematic. For open-source platforms, you can also cherry-pick only relevant updates and thus maintain your own branch of the platform. In such a case, be aware that security scanners may produce false positives and you need to be able to prove that you have patched the vulnerability.

HINT:

A security audit will also cover the underlying platforms. The auditor will check that the security features of the platform have been utilised and check for common pitfalls that the platform may have. A typical requirement for a security assessment is that all features of the platform that are non-essential to the functionality are disabled and removed, if possible. They will also want to know your update policy and that you are tracking the platform updates to keep your product secure.

SOFTWARE COMPONENTS

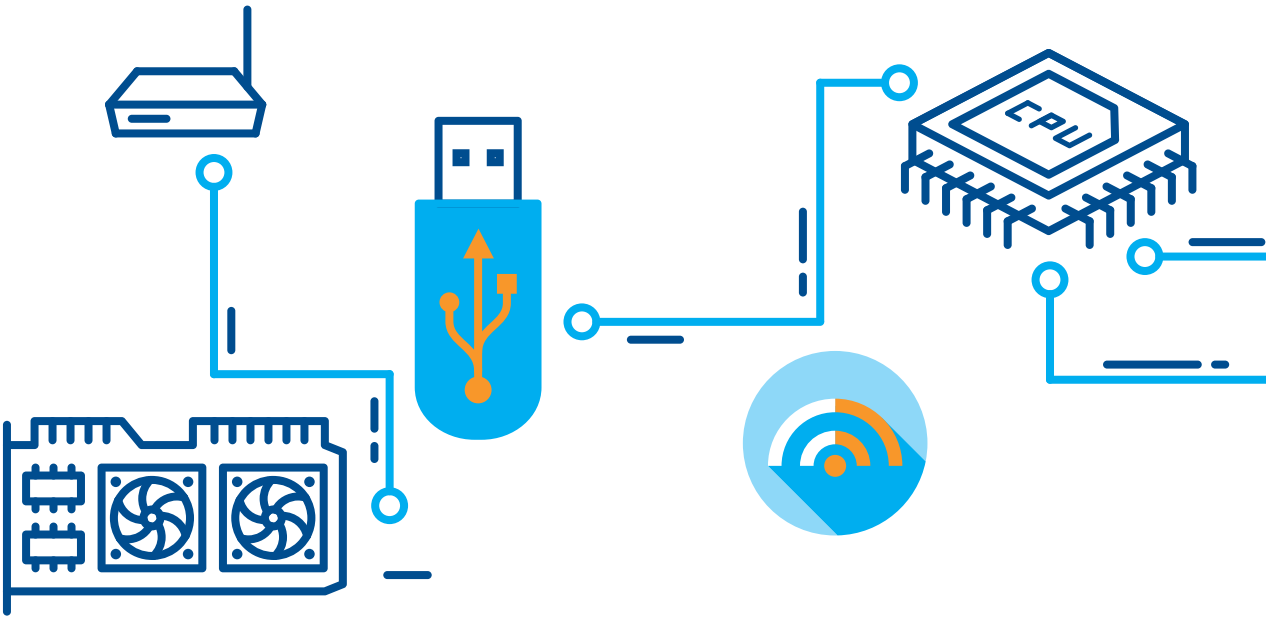
Your product has different components. You can think of these components with varying levels of abstraction: software client, backend, database, VPN terminator, source of randomness, software library and so forth. Some components you implement in-house, some are subcontracted or purchased, while some are free and others possibly open source. You need to know what components your product is actually using! Sometimes the term bill of materials is used to refer to the list of components an application is made up of.

Some components may appear more security-critical than others. Typically, components related to core business logic or security itself get special treatment. Alas, even a mundane image-handling library in high-security messaging software may be the source of a fatal vulnerability when rendering a thumbnail of the person trying to contact you. We urge you to treat all components as critical and if something really is a second class citizen in your product maybe it could be removed altogether to reduce complexity.

As a further example, installation code might not appear critical for security. However, if installer gets compromised, it could be misused to install a compromised version of your software.

Once you have the architecture and components lined up, you can think about the security characteristics of each component:

- Components use different technologies and platforms. Do you understand their impacts on your product's security?
- Different components have different expected update frequencies. How do you synchronise their updates with your product updates?
- Different components require different access rights. How do you apply the principle of least privilege to them?
- Components may have very different functionality from your core product. Do you need different testing techniques and other quality assurance measures for them?



Supply chains

Third party components form your software supply chain. The suppliers of your components are likely to use other suppliers as well, so the chain may be long. As different components are updated, the updates trickle down the supply chain, and finally you must decide which updates to apply.

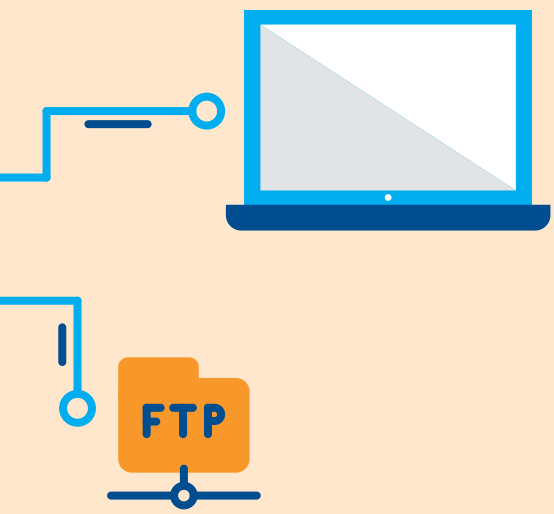
The OpenSSL cryptographic library is a good example of a popular but challenging component in the supply chain. It has been bundled with numerous embedded devices, applications, services, as well as with other components, programming languages and platforms. The authors have frequently seen products with several different versions of OpenSSL bundled inside them. OpenSSL is actively maintained, and security vulnerabilities have often been found in it, so its update frequency is quite high. If your product is using the TLS protocol or X.509 certificates, then the chances are high that you are directly or indirectly using OpenSSL.

HINT:

As part of a security audit, the auditor wants to understand your component supply chain. Be prepared to show your version of a bill of materials, and include hardware components where applicable. Auditors may use tools to cross-check the list of components you provide against the actual ones that are used by the product. Components that are old, redundant, have dubious reputations, or are otherwise exotic are likely to result in questions, which you need to be ready and able to answer.

Further reading

- *OWASP: Security by Design Principles*



SECURE PROGRAMMING

Programming turns the design into an application, which hopefully meets the original requirements. Unfortunately, it is relatively easy to introduce vulnerabilities during the programming phase. On the positive side, the problem is well recognised, and there are many resources, guides, and courses for secure programming. You must know the security posture of the platform and external components you are using. You should be aware of the impact of the programming language and other tools that you choose. Writing secure code is possible, but it is as hard as writing error-free code. Thus, it is helpful to include a security angle in your code reviews.

Your code must be well documented, modular, readable, testable and tested. This is because you must be able to maintain your code over time and perform fixes to it without compromising its security.

Your code must be well documented, modular, readable, testable and tested.

Static analysis tools aim to automatically find flaws – including security problems – by analysing the source code. Static analysis tools can help you a lot, and there are both free and commercial solutions available for most programming languages. The bad news is that many of them may report a lot of false positives, like warnings about code constructs that are not actually problems. So, you should reserve time and effort for introducing static code analysis into your product development, especially if the code base is large and has not previously been analysed.

CRYPTOGRAPHY

Virtually all systems and products require some cryptographic functions, as they must transmit or store confidential information, authenticate users and services, and so on.

Cryptography lays one cornerstone of good security. It is pivotal that you get it right. Rule number one: don't reinvent the wheel! Use well-known, high-quality cryptographic libraries in your product, and avoid home-made cryptographic primitives. Always follow standards and best practices instead of coming up with your own.

One common pitfall to avoid is using a poor source of randomness. Many cryptographic functions require genuinely cryptographically strong random numbers, and it is hard to produce them. You should carefully study reliable ways to get good random numbers on the platform you plan to use.

It is easy to make mistakes when using cryptographic functions. Many applications have failed to properly validate certificates before trusting them (see Common x509 certificate validation/creation pitfalls for examples). So, use well-known established techniques and find out about how to use them properly.

HINT:

A security audit will put a special focus on the cryptographic functions of your product. In addition to a thorough design and code review, this code will also be reviewed and debugged as it runs. The source of cryptographically secure random numbers is definitely going to be reviewed. Auditors will ask which standards or well-known implementations your crypto is based on.

The Finnish version of the cryptographic strength requirements is available from: https://www.viestintavirasto.fi/attachments/tietoturva/Kryptografiset_vahvuusvaatimukset_-_kansalliset_suojaustasot.pdf. Good generic guidance is available from: http://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf. More specific advice for your use cases is available directly from NCSA-FI, please ask.

MANAGE DEPENDENCIES

Modern software development is often more about assembling components than writing proprietary software. Programming environments have developed in a direction where it is easy for programmers to import third-party components into products.

There is a huge amount of free, open-source components available for all major platforms and programming languages. Cryptographic functions, parsing data in various formats, and integration between systems are usually best handled by importing components. Even free components may have commercial support available.

You should have a policy, or an agreed process, for how components are accepted for use. The decision cannot be left to individual developers. Each component may introduce new vulnerabilities. You also need to be aware of the licenses of the third-party components you use.

Sometimes your external dependencies are just copied and pasted code. Programmers need to solve complex problems, and often a solution is available on the Internet as a code snippet. “It came from the Internet” is not a guarantee of security and may result in licensing surprises. If you don’t understand it, don’t copy it.

CONDUCT CODE REVIEWS

Reviews are excellent for ensuring secure programming guidelines have been followed. Taking care of other quality attributes – such as comments in source code, good naming practices, and so on – allows you to ship fixes with less work. They also ensure continuity by transferring information between team members.

HINT:

A thorough security audit includes a source code review. Source code that is not readable, documented, and properly version controlled is itself going to be a red flag. The auditors are not going to read and understand the whole source code, but based on their experience, usage scenarios for the product, the threat model, and other factors, they want to find and review portions of code that they consider critical. If those portions are hard to find because the code is messy, or even low quality, you might discover bumps in the road towards acceptance.

Code review is often not enough to tell if the software works correctly, or at all. The auditor might ask you to facilitate running and debugging the code. For example, many cryptographic functions can be used incorrectly, or not at all, and the auditor may want to walk through your code performing the function step by step. Your assistance may be required in this process, especially if you have exotic hardware or software components. Your auditor also needs to understand how you build and compile your code in order to assess the security of the build process itself.

CONTINUOUS INTEGRATION

Continuous integration (CI) means that new builds of software are made regularly. With automation, the product is rebuilt after each commit to the source code repository. You can also add automated tests to the process to provide developers with immediate feedback about potential errors they have made. With continuous integration and automated tests, errors are fixed right after they have been created, and you can be more confident about each build.

Creating a continuous integration environment is an investment that you should seriously consider. Even without a fully automated CI, we urge you to move towards the capability to create frequent builds of your product and to have as many automated tests as possible.

Further reading

- *Secure coding guides*
  - *OWASP*
  - *SEI CERT Coding Standards*
- *References for more secure compilation options in some systems*
  - *C-Based Toolchain Hardening (Microsoft and GCC)*
  - *Debian Hardening*
  - *Microsoft - Security Best Practices for C++*

TESTING AND VERIFICATION

Testing checks whether an implemented system meets its requirements. This includes written explicit requirements, but also the implicit requirements. For example, software should not crash even when this has not been explicitly expressed in the requirements.

You should also pay attention to negative tests – try things that should not succeed.

The security features of a system should be tested, as well as other important system functionalities. You should also pay attention to negative tests – try things that should not succeed. As we are looking for a high-quality system, testing automation is required, as manual testing is simply too laborious to achieve good coverage for larger systems with regular builds. To get started, let's first recap all the required testing activities.

Unit testing has automated, developer-driven tests for portions of code in a single component. As developers are expected to run the tests themselves, fixing bugs that are found should be fast and cheap. Code reviews are a great place to check that unit tests have been developed for most code.

Component testing executes a component in isolation, perhaps with other simulated components representing the whole system. Component tests may be designed by a testing team. Component tests are ideally automated and executed every day or night.

System testing exercises a build of the full system. System tests may require manual testing activities, and thus may be time-consuming and expensive compared to unit tests or component tests. It is also possible to automate system tests, but that may require significant investment in testing infrastructure.

Acceptance testing is performed by an independent testing team, a customer, or by a third party. Significant problems found during acceptance testing may lead to extensive changes and redoing the acceptance tests, which may prove to be very expensive and time-consuming.

Static testing is done without running the actual product, but by inspecting various artifacts, such as source code and binaries.

- Code reviews and inspections can be seen as a form of static testing.
- Automated source code analysis is static testing.
- One fairly new approach is software composition analysis, which takes a compiled binary and then inspects which components have been used to assemble it. This process reveals the external dependencies a product has, which can be otherwise hard to enforce when the number of developers and modules in a product grow.

Dynamic testing exercises the product and inspects its behaviour

- Traditionally, this means performing manual or automatic tests that verify conformance with product requirements.
- Fuzzing is a security-oriented dynamic testing technique.
- Today, you should also test against security requirements, and attempts to attack and abuse the product should be included in the dynamic tests.
- Load testing is a dynamic testing activity focusing on performance.

HINT:

A security auditor may do the following to test your product:

1. Read your user manual and/or ask for training on the product
2. Configure your product for their intended use
3. Use the product according to their intended usage scenario
4. Click all available menus and dialogs of the product
5. Attempt to administrate the product according to its intended use
6. Create exceptional and stressful situations for the product (more about them later)

Also, remember that the auditors will cross-correlate what they see here to the threat model and update it as required. After all, their goal is to understand what the most relevant threats are, whether they match original specifications or not.

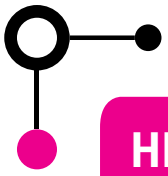
FUZZING

Fuzzing, also called fuzz testing, is a security-oriented testing method where the product is subjected to unexpected and erroneous inputs in order to find bugs.

Fuzzing can be done without access to your source code. It is likely to reveal problems, and you want to catch them before others do. Fuzzing can also be fully automated.

Fuzzing finds vulnerabilities. A typical first sign of a vulnerability is a crash or a denial-of-service condition, but it does not stop there. With specifically crafted input, an attacker can potentially take control of the system. This is called exploitation. There are various free and commercial products for fuzz testing. (See the further reading for a list.) You should use them, as attackers will.

Fuzzing is likely to reveal problems, and you want to catch them before others do.



HINT:

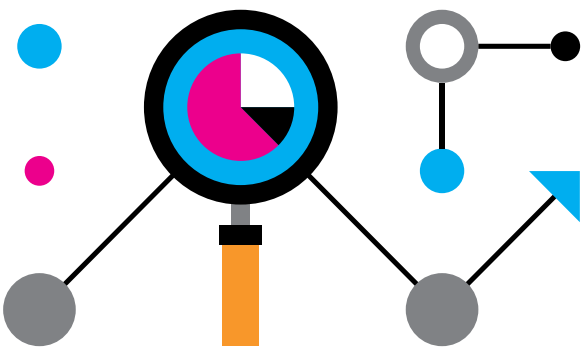
Fuzzing is often part of a security audit due to its nature: it is easy to do without knowing the specifics of the product internals and still effective at discovering flaws. Many, if not all, common platforms have been extensively fuzz-tested, so the auditor is not likely to fuzz test them any more. Fuzzing is most likely applied to your own home-made interfaces and exotic components.

PENETRATION TESTING

Penetration testing is a security testing activity performed by dedicated security experts, who try to break into a service or system and point out security problems. Penetration testers are frequently used, and they can give an impartial opinion about the security stance of the system. However, as penetration testing is manual work, it usually cannot be performed for each product version or build. It also may not be consistent on in terms of breadth and depth, and thus can be seen more like an acceptance testing activity.

STRESS OR TORTURE TESTING

An attacker may look to expose your product to unusual stress or circumstances in order to find vulnerabilities. It is often not possible to prevent this, so your system should employ aforementioned principles of "fail securely" and "defence in depth". Some scenarios that you should consider:



- What happens when a device starts up? Are there unidentified attack vectors during boot (such as a key combination to get to the system menu), or will the device accept a firmware update from anyone during boot? (This has happened.)
- What happens when a network gets disconnected? Does the system crash or enter an unsafe state?
- What happens if there is a power outage and the device reboots once power is restored?

HINT:

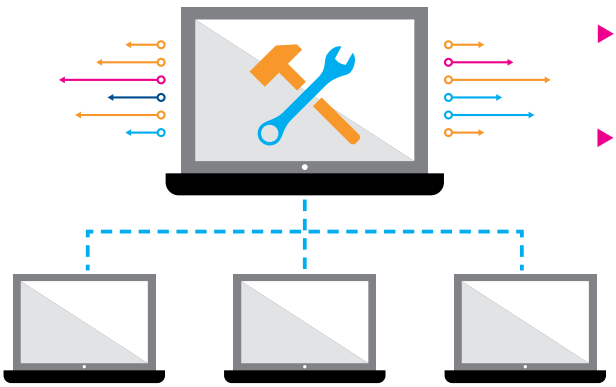
Creating stressful situations for your product is going to be part of a security audit. They are usually easy to simulate (e.g. pull the plug), yet they can reveal quite a lot.

REVERSE ENGINEERING

We said earlier that you should not base the security of your product on secret details of your program and algorithms; you should not build security by obscurity. One good reason is that there are a lot of tools to reverse engineer executables, firmware and network traffic.

It is not realistic to assume that the system engineering details will remain hidden.

Consider reverse engineering your own products, just to get a feel for it. We have found even these basic tools useful:



- **Wireshark** to sniff and analyse network traffic
- **Nmap** to scan network for hosts, open ports and services
- **Strings** to output strings from any file (e.g. executables, firmware)

HINT:

A security auditor will use reverse engineering to double-check that your claims about your product are true. Which components are actually used in the product? Do the network scan results from your product match the list of network services required? Is the network traffic actually what you claim it to be? Are the physical security components what you claim that they are?

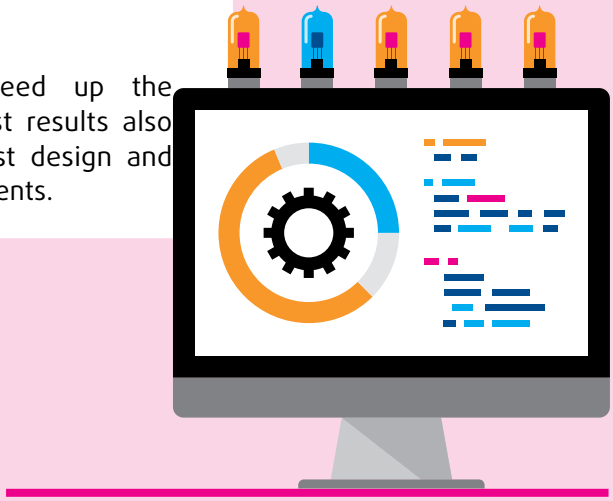
TESTING SUMMARY

We don't want to fail acceptance testing. The best way to avoid rejection is to understand the acceptance tests and then do more rigorous testing in-house before seeking acceptance. Perhaps you could use a third party to do more testing for you before providing your system for acceptance testing.

A situation where everybody has been: A system works perfectly on "my computer" or in your own lab, but fails miserably when tried elsewhere. You should definitely avoid the situation where acceptance testing is the first time your system is used outside your company. Using third-party testers is one way to avoid this. They could be motivated beta-customers, who can provide you with a real-world usage situation and feedback on your product.

HINT:

Well-documented testing will speed up the assessment process. Third-party test results also help. Remember to reflect your test design and results against your security requirements.



Further reading

- *OWASP: Web Application Security Testing Cheat Sheet*
- *Wikipedia: Penetration Testing*
- *Wikipedia: Stress testing*
- *OWASP: Fuzzing (with some tool references)*
- *MICKAEL DORIGNY Updated 19/10/2016: What is hardening*

DEPLOYMENT

Once your product is ready and someone has bought it, you need to deploy it for the customer. In the past, that involved providing an installation disk or sending someone to install the system for the customer. Nowadays, systems are installed over the Internet, or at least updated over the network after the initial installation. Cloud services are not installed at all, but used with a Web browser.

Nowadays, your application may also come in the form of a container image, such as a Docker image. Containers may include a large number of different components whose security and supply chain you need to understand. In the case of high-security products including a hardware component, you should also think about ways the hardware could be tampered with.

You should instruct your customers about secure lifecycle management of your products.

When you provide a download portal for your customers, you must ensure that nobody can infect the installation images in the portal. Moreover, you should ensure that your customer does not fall for fake download portals when they are looking to download your product. This requires that you properly secure the portal server and always use encrypted webpages with up-to-date certificates. If your product itself downloads updates or extensions, it should always check that the origin of the downloads is indeed your own server. Even with signed updates, you should beware of downgrade attacks where an attacker lures your customers to update to an older, vulnerable version of your product.

Your deployment format may be an appliance running some OS, for example Linux, or a virtual image, which is deployed by the customer. Whatever the platform is, you should make sure that it is security-hardened appropriately. This typically means disabling, and perhaps even removing, all the services that are not essential to the product. Platforms often come with their own optional security features, and you need to consider whether to enable them or not.

Possible locations are at the manufacturing site, in transit from the manufacturer to your facilities, or in transit from your facilities to the customer. At the other end of the product life cycle, you may have to think about what happens to the hardware when it is decommissioned. For example, the disks inside the product may contain sensitive data, which must be destroyed. You should instruct your customers about secure lifecycle management of your products.

MAINTENANCE AND PATCHING

Once your product has been approved, purchased, and installed, there will be maintenance. Sooner or later, a vulnerability is likely to be discovered, most likely in some third-party component you have integrated or in the platform you are using. You will have to respond quickly and diligently.

Assuming a specific version of your product has been approved, an update may invalidate that approval. However, if you do not update, your customers remain vulnerable. This is something you should prepare for and discuss with the approver. Some angles to this problem:

- Small point changes are easier to accept than overall changes with uncertainty over what the final scope of the change is.
- Changes in some system components may be less worrying for the approver compared to others. For example, updates in user interface components may be fine without new approval, whereas the cryptographic module might not be modified without requiring new approval.
- If the approver knows that you have a solid development, verification and release process, they may feel less worried that updates could lead to unwanted side effects in your products.

You definitely do not want to be in a situation where a customer asks if a recently discovered vulnerability affects your system, and you do not know if it does or not. This requires you to understand the composition of your product and your software supply chain. It is even better if you can proactively inform your customers about vulnerabilities affecting your product. This requires you to have a process to follow when vulnerabilities are found in the relevant components and platforms.

In 2014, the Heartbleed bug (CVE-2014-0160) was discovered in the OpenSSL cryptographic library. OpenSSL is very popular and is used directly and indirectly by a large number of different applications, devices, components and platforms. As Heartbleed was a nasty bug that gained a lot of attention, there was general pressure to fix the bug wherever OpenSSL was used. This caused customers to ask their vendors if OpenSSL was being used and if they might be vulnerable. Many vendors did not know which version they were using and if they were vulnerable. Some vendors did not know that they were using OpenSSL in the first place.

It is also possible that someone could find a vulnerability in your product and wants to tell you. We assume that if that person is your customer, you are eager to listen and pay attention. However, that person may also be someone else, perhaps a security researcher. In that case, you should consider having a channel, for example a specific email address or a Web form, for making these reports. If researchers cannot reach you and get feedback, they might publish the vulnerability or sell it to a dubious party. There is a good Finnish-language website, https://www.tieto-turvailmoitus.fi/, which provides basic information about how to receive security reports.

Bug bounty programs have been an effective way to improve product security. However, you need to have done your homework and have a decent maturity level to start with before you turn external parties into your only security testers. At minimum, do not sue people telling you that you have a problem. Instead thank them – or even hire them!

You are likely to bump into the following terms when you enter the world of security updates:

- CVE - Common Vulnerabilities and Exposures is a cataloguing system for identifying vulnerabilities and exposures, so that we all know that we are talking about the same thing when we talk e.g. about CVE-2014-0160.
- CVSS – the Common Vulnerability Scoring System provides a numerical "measurement" of the severity and impact of a vulnerability or exposure.
- CWE - Common Weakness Enumeration is a vocabulary for software security weaknesses. It is much less visible than the previous two.

CONCLUSIONS

This guide has covered different phases of secure development life cycle. As a self-test, we can reflect this against the Katakri auditing criteria as applied to a product vendor. Katakri would require that:

1.
The information assurance knowledge of software developers has been verified.

2.
During the software development phase, a risk analysis has been carried out and the potential risks have been dealt with (either controlled or deliberately accepted).

3.
Interfaces (at least the external ones) have been tested with false inputs and with a large quantity of inputs.

4.
Depending on the development environment, there is a policy in use for functions and interfaces that easily create problems, and this policy is monitored (e.g. Microsoft has lists of denied functions).

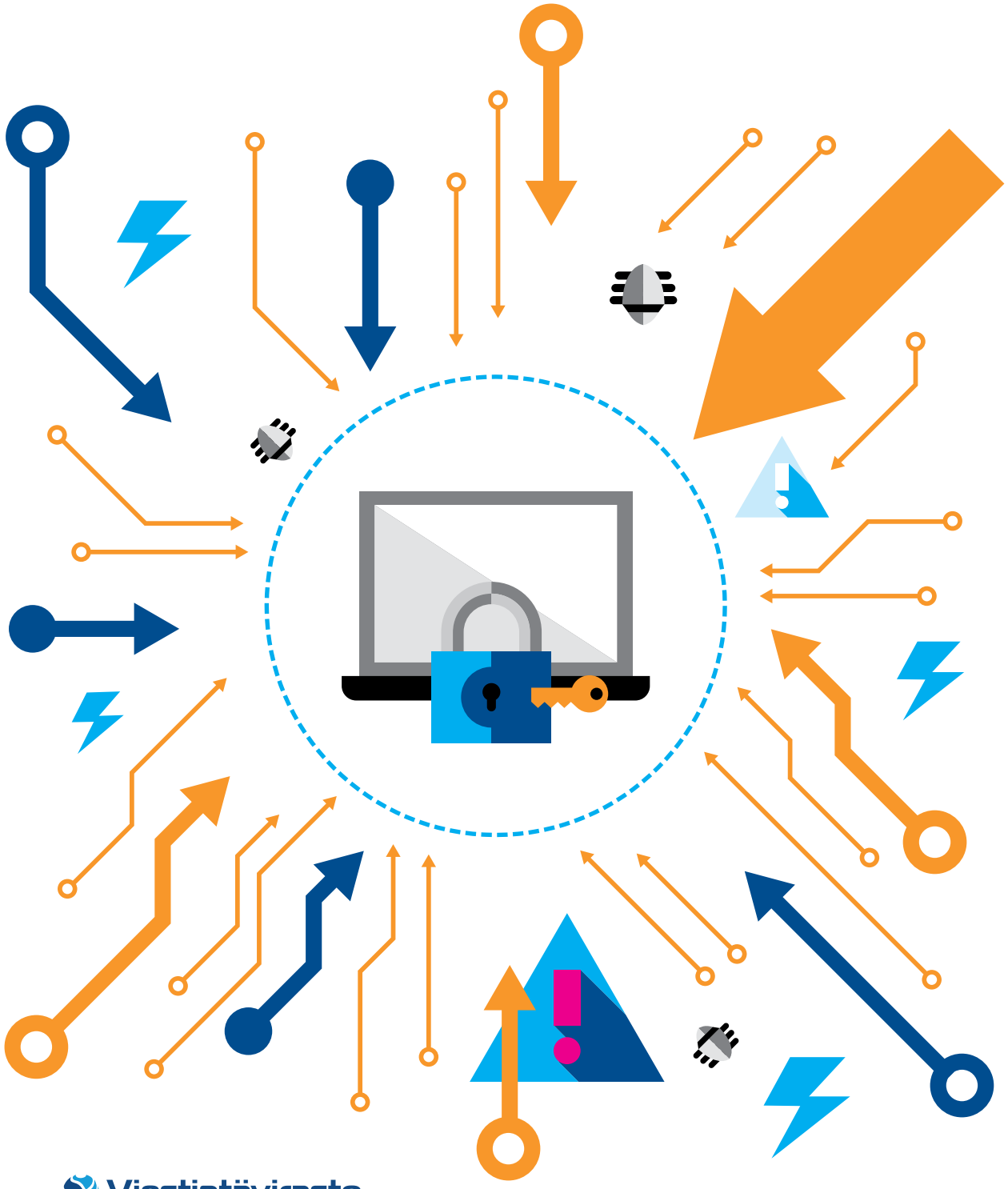
5.
The architecture and source code are audited.

6.
The product's source code is inspected with automated static analysis.

7.
The integrity of the product's source code, its version management and the development tools used is ensured.

Further reading

- *NCSA Documents (in Finnish)*
- *Katakri 2015 - Tietoturvallisuuden auditointityökalu viranomaisille - English translation*
- *VAHTI 1/2013 Sovelluskehityksen tietoturvaohje*
- *OWASP, which describes itself as follows: "OWASP is an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted." They have produced a lot of stuff, so much that it gets confusing. A couple of picks:*
  - *OWASP / Author: Dharmesh M Mehta: Effective Software Security Management*
  - *Development OWASP Guide 3.0*
- *Microsoft SDLC was one of the first published secure development life cycles.*
- *NIST - Cryptographic Standards and Guidelines*



Viestintävirasto
Finnish Communications Regulatory Authority

www.ncsc.fi | www.ficora.fi