Programming with Security in Mind

Presenter:

Mladen Marev

Senior Tech Lead, Barclays

mladen.marev@gmail.com

www.linkedin.com/in/mladenmarev

За въпроси:

Сайт: https://www.sli.do

Код: **#security-in-mind**

Аз:

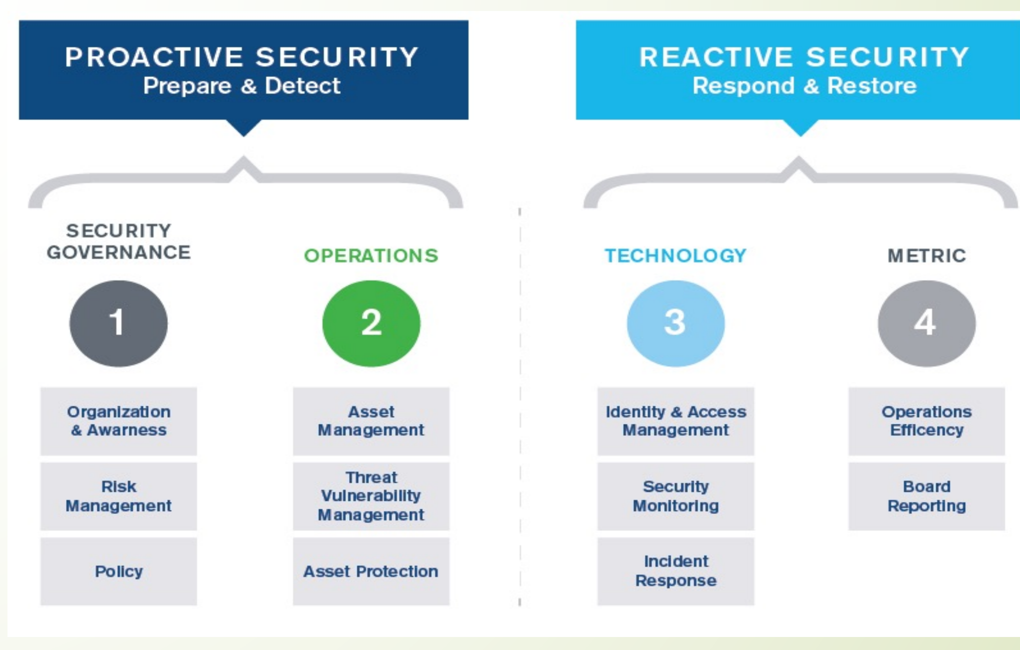
- Повече от 20 години в Софтуер Девелъпмънт с фокус в Cyber Security – PKI, Authentication and Authorization, SSO and MFA, Vulnerability management, др.
- Програмист на Java, Python, C/C++, etc.; DevOps и DevSecOps (SecDevOps)
- Security and Solution Architecture.
- Женен, с дъщеря.

Съдържание

- Cyber Security в Development lifecycle
- Стандарти за сигурно програмиране
- Управление на уязвимости
- SAST, DAST, composition analysis, pen testing, др.
- Различни скенери на пазара
- Q&A

Cyber Security в Development lifecycle

- "Well set up" компания, за да посрещне успешно предизвикателствата в сферата на кибер сигурността – основни положения:
  - Независимо от прилаганият модел на управление – Waterfall, Agile, DevOps, компанията се нуждае от добре структурирани и прецизирани политики и процедури по защитата.
  - CIO, CISO и CSO; Physical security, Network Security, Risk Management, IAM, Assessment и Testing, Operations, Software Development;
  - Проактивна and реактивна защита
    - Еднакво важни
    - Изискват правилните ресурси
    - Описват ясно границите за защита
- Контрол и проверка при всички точки
  - Регулярни упражнения/проверки
  - Всяко следващо упражнение да надгражда предходното



6

Cyber Security в Development lifecycle

Исторически поглед – Waterfall:

Training	Requirements	Design	Implementation	Verification	Release	Response
	2. Establish Security Requirements	5. Establish Design Requirements	8. Use Approved Tools	11. Perform Dynamic Analysis	14. Create an Incident Response Plan	
1. Core Security Training	3. Create Quality Gates/Bug Bars	6. Perform Attack Surface Analysis/ Reduction	9. Deprecate Unsafe Functions	12. Perform Fuzz Testing	15. Conduct Final Security Review	Execute Incident Response Plan
	4. Perform Security and Privacy Risk Assessments	7. Use Threat Modeling	10. Perform Static Analysis	13. Conduct Attack Surface Review	16. Certify Release and Archive	

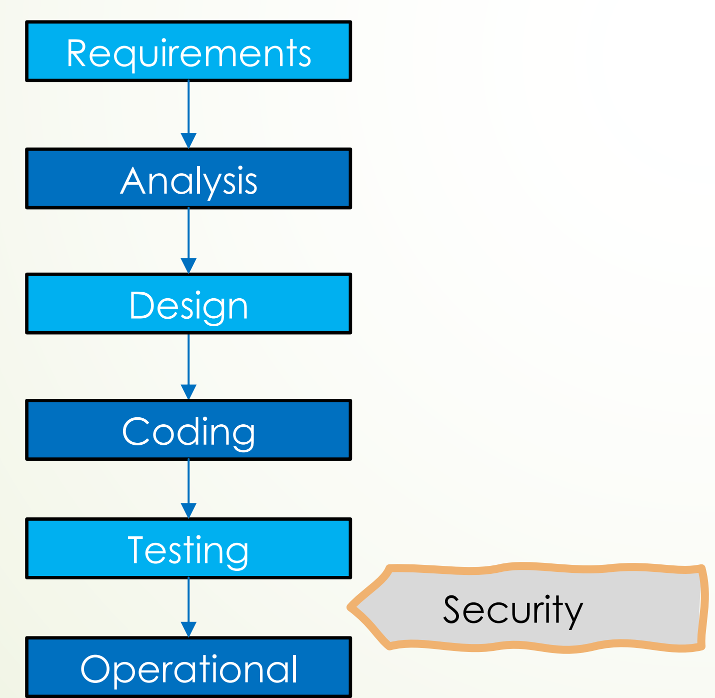
Agile Manifesto:

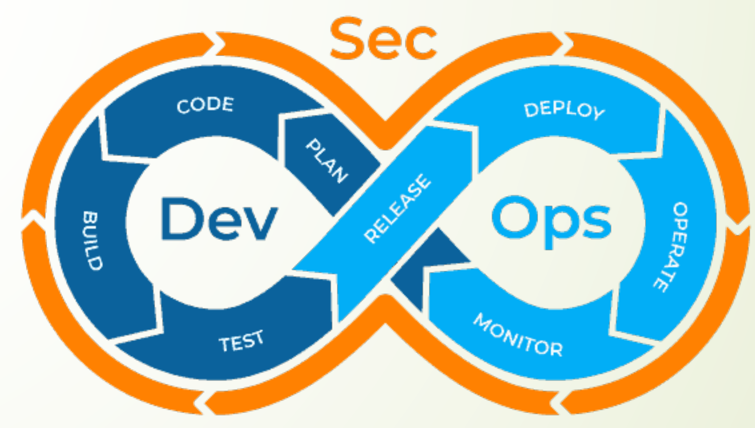
Cyber Security в Development lifecycle

Какво е DevSecOps?

DevSecOps = Development + Security + Operations

Преди & Сега





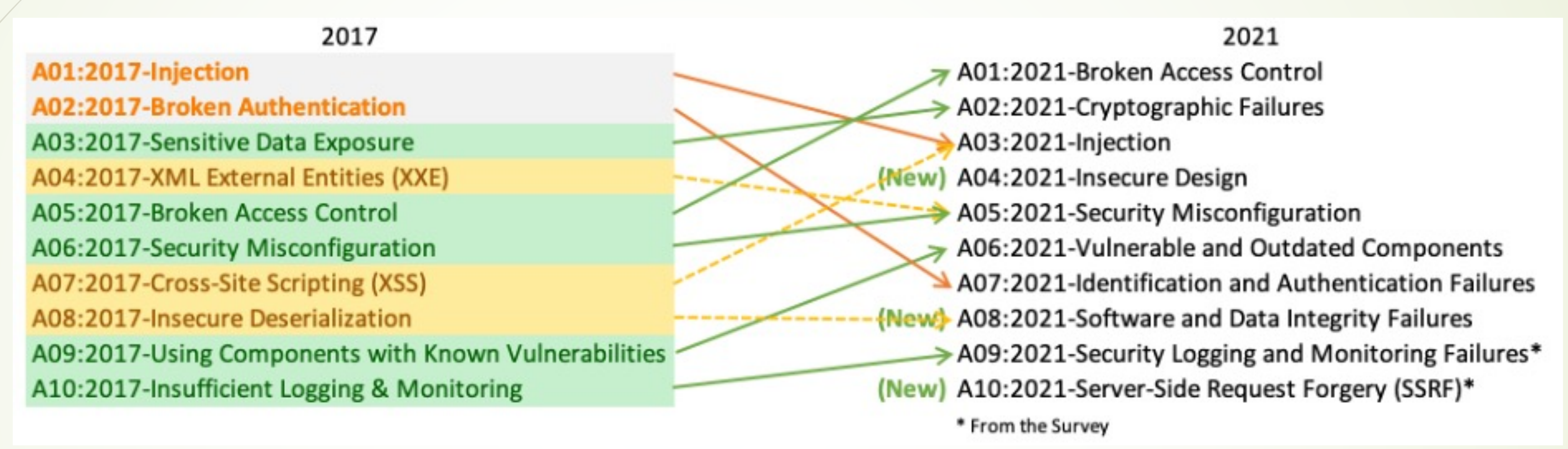
Стандарти за сигурно програмиране

- Сигурен софтуер – софтуер, създаден така, че да работи нормално, преди, по време и след злонамерена атака.
- Сигурен софтуер е различно от софтуер за сигурност – писането на сигурен софтуер е нещо, което всеки разработчик на софтуер трябва да следва.
- Стандартите за сигурен софтуер са правила и насоки за избягване на уязвимости. Използвани и прилагани правилно, може да се избегнат, засекат и елиминират проблеми, които могат да компрометират системата.
- Основни правила
  - Управление на достъпа (Managing Access Control) - принцип на „Най-малко привилегии“ (Least Privilege), „Разделяне на задълженията“ (Separation of Duties)
  - Осигуряване на защита на данните (Data Protection) – минимизиране на площта на атаката, услуги „No Trust“.
  - Защита на уязвимостите в Кибер сигурността
- Ключови стандарти:
  - CWE (Top 25)
  - CERT
  - OWASP (Top 10)

Стандарти за сигурно програмиране – OWASP top 10 2021

- A01:2021- **Broken Access Control** – над 94% от системите се тестват за проблеми в контрола за достъп. 34 от CWE уязвимости са свързани с това, и те са най-често срещаните проблеми в системите.
- A02:2021- **Cryptographic Failures** – известни преди това като Sensitive Data Exposure Обикновено са съпътстващи уязвимости, а не в основата на атаките. Ре-фокусирането тук се дължи на все повече уязвимости, свързани с достъп до данни, базирани на крипто проблеми.
- A03:2021- **Injection** – може би най-популярната уязвимост, като от 2021 тук е сложено и Cross-site Scripting, 33 от CWE уязвимости са свързани със скрипт инжекциите.
- A04:2021- **Insecure Design** чисто нова категория от 2021, като фокуса е в дизайн уязвимости. Обхваща threat modelling, secure design patterns и principles, и reference architectures.
- A05:2021- **Security Misconfiguration** – над 90% от системите се тестват за проблеми в конфигурацията. С прехода в облачни услуги, микро сървиси, конфигурациите стават все по важни и критични. Тук се включва вече и XML External Entities (XXE).
- A06:2021- **Vulnerable and Outdated Components** познато преди като Using Components with Known Vulnerabilities и е на второ място в Top 10 на най-често допусканите уязвимости. Използването на стари, дори излезли от поддръжка компоненти, прави системата уязвима, дори и „нашият“ код да е „най-добрият и чист“.
- A07:2021- **Identification and Authentication Failures** известно преди като Broken Authentication и бележи лек спад. Причината е най-вече в използването на централизирани системи в Cloud, както и в изчистването на стандартите, фокуса върху паролите и МФА.
- A08:2021- **Software and Data Integrity Failures** нова категория от 2021, като фокуса е към периодичните обновявания на системите, критичните данни, и CI/CD pipelines без проверка на интегритета.
- A09:2021- **Security Logging and Monitoring Failures** позната преди като Insufficient Logging & Monitoring. Пробив в тази категория, води до проблеми във видимостта, следенето за инциденти, откриване на пробиви и мониторинга.
- A10:2021- **Server-Side Request Forgery** новодобавена категория, по настояване на бизнеса. Въпреки доста ниската вероятност за използване, при атака по тази категория, щетите може да са големи. Това е уязвимост в уеб защитата, която позволява на нападателя да индуцира приложението от страна на сървъра да прави HTTP заявки към произволен домейн по избор на нападателя.

Стандарти за сигурно програмиране – OWASP top 10



- Подреждането постоянно се променя, също както и самите системи
- Облачните услуги поставят нови предизвикателства и са основен мотиватор за промяната на фокуса при изследвания на уязвимости
- Важността на проблеми, свързани с средата (особено „on premise“) намалява
- Забелязва се скок на проблемите и уязвимостите, свързани с автоматизацията на процесите

Стандарти за сигурно програмиране – правила/препоръки

- Input Validation and Data Sanitization
  - Валидиране на всички данни, идващи от не-доверени източници. Правилно валидиране и проверка на тези данни може да елиминира голяма част от уязвимостите. Бъдете подозрителни към всеки източник на данни, включително аргументи от конзола, мрежови интерфейси, променливи на средата и „собствени“ файлове. В последно време, ВСИЧКИ източници на данни се приемат за подозрителни – вътрешни и защитени бази данни, конфигурационни файлове и др.
  - Почистване на всички данни, предавани или запазвани в комплексни системи. При атака, може да се атакува всеки компонент на системата и дори вече почистени данни да се инфектират при трансфер.
- Heed compiler warnings
  - Винаги, при възможност, компилиране с най-високо ниво за предупреждения (warning level) и разглеждане на всяко предупреждение.
  - Използване на системи за статичен и динамичен скан, за качество на кода, антивирусни програми, и други..

Стандарти за сигурно програмиране – правила/препоръки

- Architect and design for security policies
  - При дизайн на софтуер да се залагат контроли, архитектурни модели, които да наблягат на зашитата на системата, данните и потребителите – "security by design". Например – ако системата изисква различни привилегии по различно време, дизайна може да предвиди разделянето на системата на съб-системи, с различни привилегии.
- Keep it simple
  - Колкото по-прост е дизайна и имплементирането, толкова по-лесна за поддръжка, и съответно по-защитена е системата. Сложният дизайн увеличава вероятността от грешки по време на имплементиране. Освен това, при увеличаване на сложността на защитните механизми, интегрирането им в сложните, комплексни дизайни е много по-трудно и времеемки.
  - Писане на код колкото се може по-просто. Не-комплексният код е лесен за поддръжка, лесен за четене и разбиране. Колкото по-сложен и комплексен е кода, толкова повече уязвимости се генерират.
- Default deny
  - Достъпа по подразбиране да е забранен за всички, изключение да е достъп, а не спирането на такъв. Използване на „white list" вместо "black list". Премахване на достъпа в момента, в който не е нужен вече, или предоставяне на такъв с времеви рамки.

Стандарти за сигурно програмиране – правила/препоръки

- Adhere to the principle of least privilege
  - Подобно на предходното правило, но приложено при процеси. Процесите да се стартират с най-малкото възможни привилегии. Достъпването на ресурси от процесите да е отново с най-малкото възможни и необходими привилегии.
  - Отново white вместо black list.
- Practice defence in depth
  - Стратегия за управление на риска с повече от едно ниво на защита. Така, ако едно ниво поддаде, другите могат да отразят атаката. Казано с други думи, всяко от нивата на защита могат да предотвратят един проблем да стане уязвимост.
  - Например: комбинирането на добре написан код с добре защитена runtime система, би редуцирала риска от употреба на уязвимост в кода, защото средата е защитена. Доста от уязвимостите разчитат на вид man in the middle, но ако има двустранна TLC връзка, вероятността от употребата ин намалява.

Стандарти за сигурно програмиране – правила/препоръки

- Use effective quality assurance techniques
  - Качествен код = защитен код. Добрите техники за управление на качеството на кода могат да са ефективни и при установяване и отстраняване на уязвимости. Unit testing, regression testing, E2E scenario testing, penetration testing, и сорс код ревюта трябва да са част от ефективната програма за качество на кода. Използването на независими външни ревюта за сигурност на сорс кода е мощно, но скъпо средство за осигуряване на защитени системи.
- Adopt a secure coding standard
  - Разработва не следване на вътрешни стандарти за програмиране е ключов елемент от цялостната стратегия за защита.
- Define security requirements
  - Установяване и документиране на изискванията за защита на системата във възможно най-ранна фаза и контрол за прилагането и имплементирането им. Всеки модул и ниво на системата, не само сорс кода, трябва да са подравнени по тези изисквания. При липсата им, няма как, или е много трудно да се оцени до колко сигурна е системата.
- Model threats
  - Моделирането на заплахи е процес, чрез който потенциалните заплахи, като структурни уязвимости или липсата на подходящи предпазни мерки, могат да бъдат идентифицирани, изброени и смекчаването може да бъде приоритизиране. Това включва идентифициране на ключовите активи, декомпозиране на системата, установяване и категоризиране на заплахите за всеки актив и компонент, определяне на риска и създаване на мерки за защита.

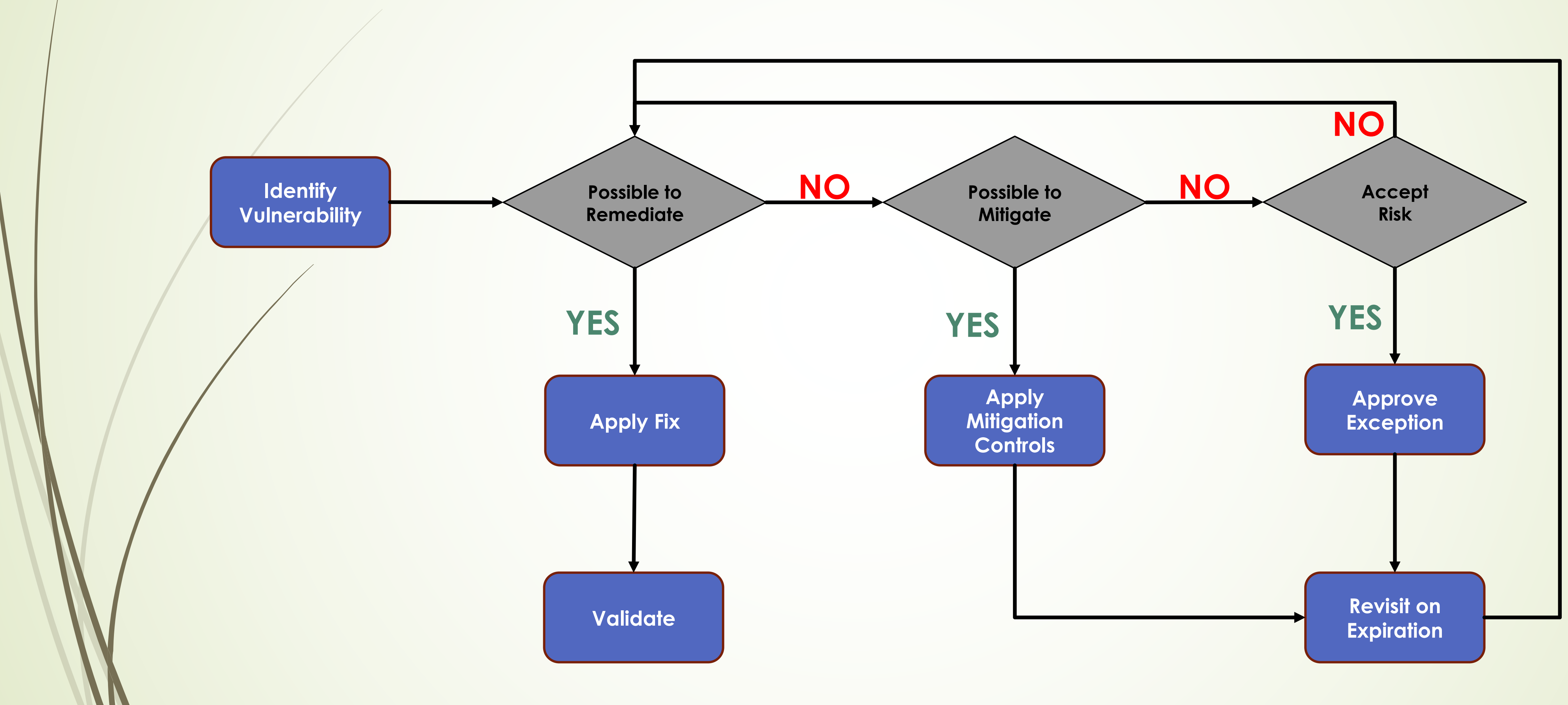
Управление на уязвимости

- **Vulnerabilities, Exploits, and Threats**
  - Уязвимости (Vulnerabilities) – това са програмните грешки, които биха могли да бъдат вредни за приложението и които използвани самостоятелно или в микс, могат да дадат предимство на някой да злоупотребява със системата.
  - Експлойти (Exploits) са средствата, чрез които уязвимостта може да бъде използвана за злонамерена дейност от хакери; те включват части от софтуер, последователности от команди или дори комплекти за експлоатация с отворен код.
  - Заплахи (Threat) се отнасят до хипотетичното събитие, при което нападателят използва уязвимостта. Самата заплаха обикновено ще включва експлоатация, тъй като това е често срещан начин хакерите да направят своя ход.
- **Уязвимости в сигурността** от своя страна се отнасят до технологични слабости, които позволяват на нападателите да компрометират продукт и информацията, която той управлява. Процес по следене и установяване на уязвимости трябва да се извършва непрекъснато, за да бъдете в крак с добавянето на нови системи или модули, промените, които се правят в системите, и откриването на нови уязвимости с течение на времето.
- **Управлението на уязвимости** е процесът на идентифициране, оценка, третиране и докладване на уязвимости в сигурността в системите и софтуера, който работи върху тях. Това, приложено заедно с други тактики за сигурност, е от жизненоважно значение за организациите, за да дадат приоритет на възможните заплахи и да сведат до минимум тяхната „повърхност на атака“.

Управление на уязвимости - lifecycle

- **Откриване (Discover)** – използват се различни видове тулове и подходи, като сканиране за уязвимости, преглед на сорс кода, тестване и др.
- **Приоритизиране (Prioritize)** – базира се на информацията от известни бази данни с уязвимости, като MITRE, CWE, OWASP.
- **Оценка (Assess)** – определя се обхвата, вероятността да бъдат използвани уязвимостите, риска, и др.
- **Докладване (Report)** – създаване на запис, свързан с уязвимостта в системата за проследяване на грешки.
- **Поправяне** или **облекчаване (Remediation** or **mitigation)** – отстраняване на проблема, ако е възможно, смекчаване/приемане на риска, ако не е.
- **Проверка (Verify)** – използват се същите инструменти, използвани за откриване на уязвимостта, за да докажете, че тя е отстранена

Управление на уязвимости



Установяване на уязвимости

- SAST - Static Application Security Testing
- DAST - Dynamic Application Security Testing
- SCA - Software Composition Analysis
- IAST - Interactive Application Security Testing
- RAST - Runtime Application Self-Protection
- Penetration Testing
- Security Code Review

19

Установяване на уязвимости - SAST

- Static application security testing (SAST), или статичен анализ на кода, е метод за изследване на сорс кода за познати уязвимости. Обикновено приложенията се сканират преди компилация.
- Познато също като **white box** testing.
- Използва се от разработчиците на приложения от вече десетилети. SAST помага за ранното откриване на потенциални уязвимости, още в фазата на разработка, като това прави остраняването бързо, лесно и евтино.
- Предимства:
  - Лесно за инсталиране, настройване и използване
  - Способност да се разшерява при промяна в мащаба
  - Поддържат се почти всички използвани в момента езици за програмиране
  - Доста нисък процес на false positives
  - Лесно се интегрира в процеса на разработка, в средите за програмиране и след това – в CI/CD средите.

Установяване на уязвимости - DAST

- Dynamic application security testing (DAST) технологиите са създадени да откриват състояния, които са потенциални уязвимости в системи, който работят в момента. Тук е голямата разлика със SAST, където приложението се тества като сорс код, не докато е в работен режим.
- Използва се основно при уеб приложения и при APIs.
- Познато като **black box** testing, сканирането се извършва без поглед в вътрешният сорс код или архитектура на приложението. Използва техники, близки до тези, използвани от хакери.
- Приложения:
  - Гъвкавост при приоритизиране, планиране и промяна на задачите при тестване, в зависимост оф нуждите на бизнеса.
  - Задълбочен анализ на всяко приложение.
  - Дава възможност за бързо разширяване на обхвата без голяма загуба на ресурси.
  - Сравнително евтино решение в сравнение с pen testing.

Установяване на уязвимости - IAST

- Interactive Application Security Testing (IAST) анализира кода за уязвимости докато приложението се тества от автоматизирани тестове, ръчно тестване, или всякакви други операции свързани с работата на приложението. Уязвимостите се докладват в реално време, което ознаюава, че не затруднява/натоварва с време, процеса за CI/CD.
- IAST работи „вътре" в приложението, което прави разликата с SAST и DAST. При това сканиране не се изследва цялото приложение, а само тези части, които се дотсигат от основното тестване.
- IAST работи най-добре когато се интегрира в QA среда с автоматични тестове.
- Предимства:
  - Бързо, без да добавя допълнително време в целият процес
  - Лесно за деплойване
  - Безболезнено интегриране в CI/CD
  - Може да се разчита на оценката не само на уязвимостта, но и да определи дали тя е изплзваема
  - Способност да идентифицира third-party и компоненти с отворен сорс, познати уязвимости, проблеми с лицензите, и много други потенциални рискове
  - Възможност лесно за разрастване и покриване на стотици хиляди HTTPS заявки
  - Напълно съвместимо с всиюки налични автоматиюни тестови среди, тестове за каюество, unit тестване и др.

Установяване на уязвимости - RAST

- Runtime Application Self-Protection (RASP) дава възможност за инспектиране на поведението на системата, но в контекста на средата, където е инсталирана. Сканирането „улавя“ всички заявки, като се проверява дали са сигурни и дали се валидират от системата.. RASP може да алармира, когато е настроен за диагностика, докато самата система продължава да работи. Освен това, може да се използва за усртановяване на потенциални атаки, когато е в настроено защитно. При атака може да спре напълно системата, или само определена операция.
- Дори RASP и IAST да имат подобни методи на работа, RASP не извършва цялостно сканиране, а вместо това се концентрира върху части на системата, като инспектира трафика и активността в модула. И двата метода за анализ (RASP и IAST) докладват атаки, когато се детектнат, но IAST го прави по време на тестване, докато RASP – по време на нормална работа.
- Предимства:
  - Достъп до кода на приложението
  - Реакция на пасивни и активни инциденти в изследваните модули
  - Възможност да се конфигурира да запосва в log, да алармира и дори да блокира атаки
  - Подръжка на много езици и платфроми
  - Независимо опериране, дори и в off-line системи
  - Покрива голямо разнообразие от уязвимости

Установяване на уязвимости - SCA

- Software composition analysis (SCA) е автоматичен процес, който намира и докладва софтуер с отворен код с състемата. При анализа на този код, се оценява дали е уязвим, дали е харед с лицензите, както и с качеството на кода.
- Основно предимство е надежността, скороста на сканиране и доверието в резултатите.
- Интегрирано в повечето от широко известните модерни сорс репозиторита.
- SCA обработва листа от библиотеки, на които разчита основната система, и оценява дали са сигурни.

Установяване на уязвимости – pen testing

- Pen testers са всъщност така наречените **етични хакери**.
- Penetration test, също известно като pen test, е симулирана атака към системата, като се откриват и после тестват уязвимости. В контекста на уеб базирани системи, penetration testing се използва също и за настроиване на *web application firewall* (WAF).
- Pen testing се използва за опит за компрометиране на всякаква система, включително APIs, frontend/backend системи, и др., да открие уязвимости в много по-кмплексни сценарии.
- Труден на автомотизиране, и доста скъп метод за оценка на сигурността на системите.

Установяване на уязвимости – security code review

- Извършва се от експерти както в сигурността, така и в програмирането.
- Много скъпо, много бавен процес, ръчен.
- Добавя нещо много важно – човешкият фактор (**common sense**).

Tools and vendors

- Synopsys Coverity
- HCL AppScan
- Snyk.io
- Appknox
- Micro Focus Fortify Static Code Analyzer
- SonarQube
- Klocwork
- Checkmarx
- CodeScan
- Embold
- Kiuwan Code Security & Insights
- Veracode Application Security Platform
- SpectralOps
- Argon CI/CD Security
- AttackFlow
- Netsparker

Vendors web sites

https://www.codescan.io/

https://embold.io/

https://www.kiuwan.com/code-analysis-qa/

https://www.veracode.com/

https://www.attackflow.com/

https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

https://www.sonarqube.org/

https://www.perforce.com/products/klocwork

https://www.checkmarx.com/

https://appknox.com/

https://snyk.io/

https://www.hcltechsw.com/appscan/home

https://spectralops.io/

https://argon.io/

https://www.netsparker.com/vulnerability-scanner/

Въпроси

Благодаря много

Използвани сайтове

https://security.berkeley.edu/secure-coding-practice-guidelines

https://owasp.org/Top10/

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

https://www.perforce.com/blog/kw/what-is-owasp-top-10

https://www.imperva.com/learn/application-security/

https://help.veracode.com/

https://www.ptsecurity.com/ww-en/analytics/knowledge-base/

https://www.synopsys.com/blogs/software-security/secure-sdlc/

https://kirkpatrickprice.com/blog/secure-coding-best-practices/