Sicurezza Informatica

Cristian Mercadante
UNIMORE | ULTIMO AGGIORNAMENTO: 18/12/2019

Lezione 1

Lunedì 16 settembre 2019

Introduzione

Sito: https://weblab.ing.unimore.it/sicurezza/ Login: SIC1920 Password: qcvan2050

Vulnerabilità, crimini, criminali e difese

Esistono tre elementi fondamentali: **minacce**, **vulnerabilità** e **danni**. Furto dati e cancellazione dati sono danni. Le persone sono delle vulnerabilità. Il malware è una minaccia. Ci sono anche le guerre, il terrorismo, spionaggio, fenomeni atmosferici, guasti hardware, blackout. Contro le minacce non possiamo fare nulla. Tuttavia, non tutte le minacce ci preoccupano a tutti i livelli. Ci possiamo quindi concentrare sulle vulnerabilità. Dobbiamo conoscere e diminuire le vulnerabilità della nostra azienda ed organizzazione. Non combattiamo le minacce, ma combattiamo le vulnerabilità, per diminuirle. In queste troviamo le persone, ma anche le tecnologie: non possiamo eliminare guasti hardware, ma possiamo attraverso la replicazione ridurre la vulnerabilità. Abbiamo imparato a comprendere che i nostri servizi devono essere sempre disponibili. I nostri nemici sono sempre **single point of failure** e **colli di bottiglia**. Impareremo che non tutte le minacce sono equiprobabili. Inoltre, tutte le aziende hanno le stesse minacce, ma con diverse probabilità: qual è il mio business, dove lavoro? Non possiamo utilizzare la stessa analisi di vulnerabilità per due aziende. Per ovviare alle vulnerabilità servono competenze, tecnologie, sensibilizzazione, procedure, ma soprattutto i soldi. Il bravo competente tecnico è quello che sa stabilire le **priorità**, ovvero come investire meglio i soldi. Ovviamente non possiamo avere la certezza dell'arrivo delle minacce, ma possiamo analizzare e comprendere le vulnerabilità maggiori per me, per la mia azienda. Chi sono i cattivi? Quali sono le spinte motivazionali per attaccare? Soldi, fama, paura o motivi mentali. Tra i professionisti includiamo criminali, aziende private, singole persone (*contractor*, mercenari), stati (difesa ed intelligence, che si servono di contractor, aziende specializzate, ma anche di criminali) e aziende di raccolta dati (*data broker*). Che tipo di danni può subire un'azienda? Furto di informazioni, perdita di reputazione, danni all'operatività, truffe e furti di denaro.

Gli **hacker** erano persone che si muovevano per motivi di curiosità, di sfida, senza motivazioni economiche o di barriera. Gli **hacktivist** sono quelli che si muovono per ideologie, quindi motivazioni mentali. Poi abbiamo i vandali, ovvero i **cracker**, o **black hat**, senza ritorno economico. Poi ci sono i **dipendenti interni**, spesso trascurati, spesso mossi da vendetta. Negli anni 90, la maggior parte degli attacchi avveniva di sera o nel weekend, segno che gli attacchi non erano l'attività primaria delle persone. Hanno poi pensato a far soldi, quindi questi professionisti si sono dati al ***cybercrime***: guadagno e spionaggio industriale. Si è passati poi al controllo sociale, ad esempio il sapere dove siano le persone: se queste aziende sono vicine allo Stato, è difficile scappare.

Perché tutto questo è possibile? Perché abbiamo così tante vulnerabilità? C'è un problema di **inconsapevolezza** (soprattutto C-level, ovvero nel management, CEO, CFO, COO, ecc.). **Il software fa schifo**, causa tempi e costi. Fortunatamente l'avvento dell'Industria 4.0 ha fatto sì che gli investimenti siano maggiori. Gli utenti sono **incompetenti**, ma anche gli amministratori. Servono tantissimi informatici per creare sistemi e renderli sicuri, ma tutti questi informatici non ci sono, quindi le aziende prendono chi capita.

Lezione 2

Giovedì 19 settembre 2019

Vulnerabilità delle applicazioni (1)

Abbiamo descritto i rischi che corre un'azienda. Ci sono minacce, che possono essere endogene, esogene dell'azienda. Le minacce ci sono, evolveranno ed aumenteranno. Le minacce non possono avere effetti a meno che non ci siano delle vulnerabilità all'interno dell'azienda. Potremo solo lavorare sulle vulnerabilità, che possono essere tecnologiche, umane, dove si intende sia l'incompetenza che la superficialità. Fra il 95 e 99% degli attacchi passano attraverso le persone. Questo rende il nostro lavoro estremamente complesso. La combinazione di queste minacce con le vulnerabilità crea dei danni: economici (truffe, furti dati), reputazione, operabilità.

Oggi parliamo di attacchi mirati, che cercano di sfruttare vulnerabilità aziendali a livello tecnologico. La tecnologia più vulnerabile è il software. Tolta la parte hardware, è tutto software: OS, protocolli di rete, processi server e applicazioni. Tutto il SW ha bachi (ma non necessariamente ha vulnerabilità di sicurezza). I bachi che interessano gli attaccanti sono errori software che ti consentono l'accesso anche se non consentito e che ti danno accesso ad operazioni che non potresti compiere. Il momento cruciale in cui si attacca è la fase di inserimento dell'input, perché si lascia all'utente la possibilità di agire. Quindi la maggior parte dei controlli

dovrà essere fatta all'input, ma quest'operazione è molto più lunga e dispendiosa. È impossibile verificare la correttezza assoluta del codice. Si usa spesso in informatica il rilascio di una patch, ma spesso i clienti non installano l'aggiornamento o lo fanno molto in ritardo. Chi programma dimostra pigrizia: l'uomo commette errori, si dimentica di correggerli, oppure semplicemente non sa programmare bene.

L'attaccante può sfruttare diverse metodologie per analizzare le vulnerabilità di un servizio. Il problema fondamentale è il momento dell'input, che va fatto da persone competenti, sia sulla parte di UI, sia sulla parte applicativa. Un errore tipico è la **mediazione incompleta**: fornisco delle credenziali dopo aver effettuato il controllo, ma poi mi fido di chi invece mi attaccherà. Bisognerebbe migliorare il **tempo di controllo** e il **tempo di utilizzo**. Le tre categorie di **code-injection** che analizzeremo sono il **buffer overflow**, l'**SQL injection** e il **cross site scripting**. Sono attacchi SW che sfruttano la dimensione del buffer non controllato, formattazione della stringa non controllata e accettazione di stringhe che possono causare problemi (accetto all'istante t, ma non controllo all'istante t+1, quando è cambiata). La code-injection veniva utilizzata per mandare in crash il sistema. Può essere usata per fare il *web defacement*: sulla prima pagina di un sito web faccio comparire qualsiasi cosa. Oppure posso ottenere permessi di super-utente in un sistema, installare Trojan, rubare cookies e sessioni.

Cross-Site Scripting – XSS

Parliamo di siti web, tecnologie web (http server, http client), scripting (codice come JavaScript che viene eseguito anche se non vuoi). Ci sono più entità coinvolte, per cui la parola *cross*. Il problema sono gli script tag nelle pagine web, perché io gli lascio eseguire del codice. Nasce in un contesto JavaScript, ma si è diffuso anche fra altri linguaggi. Esistono due tipologie di XSS:

- **Reflected XSS**: ci sono più entità coinvolte, c'è la complicità dell'utente, c'è la complicità di chi accetta input senza verificare che ci sia del codice eseguibile (anche non malevolo).
- **Persistent/Stored XSS**: in questo caso l'utente non viene indotto a cliccare, ma l'attaccante distribuisce delle mine su un sito che mi consente tale operazione, e chi mette il piede sulla mina, si fa male. Attenzione: non solo il sito web può essere vulnerabile, ma anche la piattaforma su cui gira, che deve essere aggiornata.

Nel primo caso, l'obiettivo dell'attaccante non è modificare nulla della piattaforma web o il contenuto del sito web. Il reflected usa altri meccanismi, con la complicità dell'utente. Nel secondo caso, l'obiettivo dell'attaccante è aggiungere qualcosa nel sito web e non richiede necessariamente il coordinamento con l'utente. Sempre nel secondo caso, come posso aggiungere contenuti al sito web? Il caso più semplice è quello in cui il sito me lo consente. Se dovessimo scrivere siti che consentano di aggiungere commenti, dobbiamo starci attenti. Se non è possibile aggiungere dei contenuti, allora si sfruttano delle vulnerabilità.

Il **reflected** fa una prima analisi di contesto per cercare siti web che sono vulnerabili. Oggi ci sono tanti strumenti semi-automatici che scannerizzano i siti web alla ricerca di queste vulnerabilità. Quando ne trovo uno, preparo un regalino per indurre uno o più utenti ad andare su quel sito web. Preparo un programmino per rubare le credenziali di utenti, installare malware. Un'attaccante cerca siti web, tipicamente con tre layer. Cerca di capire chi è vulnerabile al XSS. Gli manda delle stringhe, degli script, delle get, prova a modificare la query string. Dopodiché l'obiettivo è mandare una mail, un tweet, qualsiasi cosa che contiene sia l'hostname, sia una stringa che possa sfruttare la vulnerabilità di quel sito. L'obiettivo di questo messaggio è quello di far mandare l'utente su quel sito. L'utente ci va, viene eseguito lo script malevolo che può servire per rubare cookie, sessioni, ma anche iniettare dei Trojan nel computer del client. A quel punto con le credenziali, l'attaccante può fare operazioni sull'application server.

Nessun informatico si può permettere di non essere paranoico
M. Colajanni

Nel caso **stored**, tutti quelli che visitano quel sito sono a rischio. Nulla vieta che io possa indurre degli utenti ad andare su quel sito. Si riesce a fare questo perché non ci sono verifiche adeguate, perché ci sono siti vulnerabili, perché mi autentico in maniera inconsapevole su siti vulnerabili. L'obiettivo è quello di far eseguire un programma (ad es. un cryptominer) sui client.

Come possiamo prevenire questi casi? Sanitizzare l'input, ovvero scrivendo codice per la validazione. Dopodiché si può lavorare a livello di server http: non comunichi gli errori al client, ma solo all'amministratore. Controlli le sessioni, se ce ne sono due attive in contemporanea. Oppure si lavora a livello di codice del sito web: non visualizzi le credenziali, fai scadere la sessione. In particolare, per lo stored, facciamo girare noi degli script che verifichino che i contenuti della pagina non siano stati modificati, se non da noi: abbiamo un hash della pagina e ne calcolo il nuovo. Traggo le conseguenze dal confronto.

Lezione 3

Lunedì 23 settembre 2019

Vulnerabilità delle applicazioni (2)

L'insicurezza informatica è prevalentemente un problema del software. Tecnologie estremamente diffuse diventano un obiettivo per gli attaccanti, perché più una vulnerabilità è diffusa, più è alto il guadagno. Windows ha più minacce, perché è più popolare, quindi è più a rischio. Quindi se investo in un malware, cerco la massa del 95% degli utenti che usa windows, dove trovo l'utente che cade in trappola.

SQL injection

I database, che una volta erano accessibili solo da utenti dell'azienda, oggi sono spesso backend che si interfacciano con qualche application server, che si interfacciano con un qualche http server. I DB oggi sono estremamente più a rischio, perché le minacce che ci sono nel momento in cui il servizio è in rete sono ordini di grandezza superiori. Guarderemo l'**SQL injection**, che è diffuso ed interessante, ma approfondiremo solo la superficie. I DB sono la cassaforte di un'azienda, perché altrimenti non funziona: tutti i processi aziendali si basano su dati informatici. Questo attacco sfrutta l'SQL facendo delle query particolari con delle sintassi particolari e sfruttando un linguaggio non nato per proteggersi da attacchi, quindi un linguaggio abbastanza debole. Questo attacco si usa per entrare senza credenziali, visualizzare dati protetti, modificare e cancellate dati senza autorizzazione. L'attacco è indipendente dal DBMS, perché riguarda la programmazione: come lo utilizzi, che query accetti, ecc. Ci devo stare sempre attento, ancor di più se il database è un backend di qualche sito web.

Si sfruttano delle peculiarità della sintassi SQL per ottenere dei vantaggi. Si possono aggiungere caratteri di commento, oppure delle tautologie (si aggiunge un OR seguito da un'affermazione vera: le query ritorneranno sempre vero), oppure posso aggiungere dei comandi (*; DROP TABLE users*). Il concetto è conoscere bene la sintassi e capire come sfruttarla per fare operazioni che non dovrebbero essere previste. Il primo obiettivo è entrare, ottenere i privilegi fino al superutente e poi fare danni. L'autenticazione è interessante perché tipicamente è la prima interfaccia a cui tu vieni esposto dall'esterno. Se fai una query in cerca di corrispondenze username-password, dopo puoi accedere al sito. Oppure poi inserire un commento per evitare il controllo della password. Oppure inserire una tautologia sempre allo stesso fine. Altri tentativi meno sofisticati sono quelli di forza bruta: si provano diversi tipi di stringhe, ipotizzando che tu mi consenti di fare operazioni di forza bruta, e non mi cacci dopo 3 tentativi falliti. Quella è un'operazione da sistemista.

Buffer Overflow

Era la vulnerabilità principe del software anni 90/2000. Oggi per merito degli OS è molto più difficile. Può riguardare diversi ambiti: noi vedremo solo l'ambito dello stack segment. Più sofisticato è quello dell'heap segment. L'idea è quella di andare a scrivere oltre le dimensioni di un array, per poter fare code injection, ovvero far saltare la funzione in un punto esatto per fare eseguire del codice arbitrario. Dal punto di vista informatico noi gestiamo solo binari, quindi a chi carica in memoria i dati non cambia nulla, ma cambia a chi li deve interpretare. L'idea è scrivere nello spazio dedicato all'array dei dati che vanno oltre l'array stesso. È evidente che potrei controllare in fase di compilazione alcuni errori. Le cose diventano molto più complesse se gli errori, anziché manifestarsi a tempo di programmazione, si manifestano a tempo di esecuzione. Il programmatore deve preoccuparsene.

Ogni programma in C (che quando va in esecuzione diventa un processo) suddivide la memoria centrale in 4 parti: il testo (codice da eseguire), i dati (variabili statiche globali, tutte le strutture dati dichiarate al di fuori delle funzioni), la memoria heap (strutture dati dinamiche, dichiarate con una *malloc*) e lo stack (funzioni). Lo stack è gestito come una pila, con metodologia LIFO. Quando chiamo una funzione, alloco un frame di memoria dedicato a quella funzione che corrisponde al record di attivazione, che viene pushato quando viene chiamata la funzione e viene fatto il pop quando la funzione completa la sua esecuzione. Per fare questo serve un indirizzo di ritorno che ci dice qual è la successiva istruzione da eseguire. Così come serve un puntatore all'inizio del record di attivazione. Così mantengo una catena dove gli indirizzi sono tutti relativi al frame di riferimento. Il processore prende l'instruction pointer, che sa qual è l'istruzione da eseguire, vede qual è la lunghezza dell'istruzione, la esegue. L'istruzione può o meno cambiare l'instruction pointer. Dopo di che si ricomincia. Ciascun frame è dotato di informazioni che vengono messe in memoria principale: i parametri della funzione, le variabili locali della funzione, le informazioni per ritornare al chiamante, le informazioni per sapere dove riprendere. I registri sono lo stack pointer, il frame pointer e l'instruction pointer. Con tutte queste informazioni, l'OS può muoversi in maniera sempre corretta.

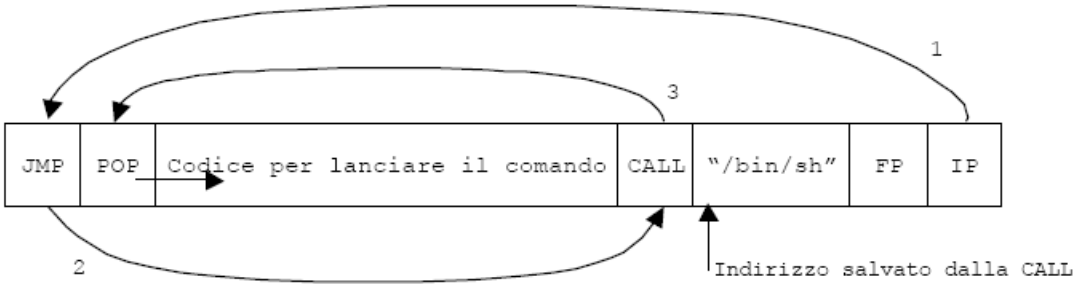
Il buffer overflow ha come obiettivo far saltare l'esecuzione in zone della memoria dove sono riuscito ad iniettare del codice malevolo. Può funzionare tenendo conto che un frame memorizza nella ram nell'ordine: variabili locali, frame pointer, instruction pointer e parametri. Se nella copia su variabili locali sfora il suo spazio sullo stack, allora va a sovrascrivere l'instruction pointer.

Se il nuovo puntatore non è un indirizzo valido, allora viene generato un *segmentation fault*. Un bravo attaccante inserisce un *jump* a una zona di codice dove c'è codice eseguibile. L'obiettivo è capire come fare **bytecode injection**. Potrei cercare di far eseguire una shell per poter prendere il controllo e poi fare un'escalation dei privilegi. Tipicamente cerco di attaccare processi che hanno già dei privilegi, quindi le shell che ne nascono hanno gli stessi diritti.

Dove metto lo *shellcode* che voglio eseguire? C'è spazio per metterlo tutto? Tipicamente no, quindi ci sono vari *jump* da una parte all'altra, ed eseguo queste operazioni che possono far perdere controllo. I sistemi operativi si sono difesi limitando la memoria disponibile e randomizzando lo spazio di memoria fornito. Per poter eseguire qualcos'altro devo decidere che *bytecode* eseguire, dove andare a metterlo e farlo eseguire. È difficile farlo se non si usano dei tool perché bisogna conoscere molto bene il C, la shell, gli OS. Devo creare dello *shellcode* di exploit. La sequenza di caratteri e di byte che ci mettiamo è molto complessa. Per quanto riguarda l'indirizzo, ce lo dobbiamo ricavare. Una modalità utile per procedere è l'utilizzo di indirizzi relativi.

Lo *shellcode* è una sequenza di operazioni in esadecimale molto complessa. Per eseguire questo codice è necessario inserire nell'instruction pointer l'inizio del buffer nel quale è memorizzato. È molto difficile perché la memoria è allocata dall'OS in modo random. È opportuno utilizzare riferimenti relativi, in modo che il programma sia in grado di calcolare da solo gli offset e quindi di funzionare indipendentemente da dove verrà allocato. Si usano le istruzioni JMP e CALL che consentono di saltare non solo a una posizione assoluta, ma anche di un determinato offset positivo o negativo a partire dall'instruction pointer corrente. La CALL salva nello stack l'indirizzo assoluto successivo a quello che la contiene, per poi garantire che l'esecuzione del programma prosegua una volta terminata la chiamata: l'indirizzo della chiamata successiva verrà quindi salvato sullo stack ed è recuperabile facendo una POP. La CALL però sposta l'instruction pointer a un'altra parte della memoria, quindi si rischia di non essere più in grado di tornare indietro, ma per non perdere il controllo la soluzione più semplice consiste nel restare all'interno del buffer (di cui si conosce la struttura e quindi tutti gli offset relativi). Di conseguenza si fa puntare la CALL quasi all'inizio del buffer. Diciamo "quasi" perché all'inizio del buffer si inserisce una JMP che punta alla CALL, posizionando quest'ultima verso la fine del codice, ma subito prima della stringa che termina il buffer. Nota bene: i reindirizzamenti interni al buffer sono facilmente calcolabili, dato che il buffer è creato proprio dall'attacker. Quindi poi:

1. Se si riesce a deviare l'instruction pointer, si arriva alla JMP.
2. La JMP porta alla CALL (è possibile sapere dove si trova la CALL perché all'interno dello stesso buffer si conosce l'offset che bisogna usare).
3. La CALL salva l'indirizzo successivo ad essa in cima allo stack e dirotta l'instruction pointer all'indirizzo dell'istruzione successiva a quella che contiene la JMP.
4. La POP recupera l'indirizzo in cima allo stack (è l'indirizzo del comando che vogliamo eseguire) e lo mette in un registro da cui è possibile recuperarlo.
5. Ora si può iniziare con il codice vero e proprio per sistemare tutti i parametri e gli offset per consentire l'esecuzione del comando.



Ci si semplifica la vista inondando di *No Operation* l'area prima del *jump*, così ho un margine di errore sull'indirizzo e posso slittare. Oppure si ripete l'indirizzo di ritorno dopo lo shell code. È possibile causare buffer overflow anche sull'heap, oppure usando le funzioni di formattazione delle stringhe.

Mediazione incompleta, tempo di controllo e tempo di utilizzo

Parliamo di programmi che non verificano al momento giusto che si abbiano o meno le autorizzazioni per compiere certe operazioni. La mediazione incompleta è un problema che si ha nel momento in cui si inseriscono i dati e che alcuni input hanno come conseguenza della sensibilità sull'autenticazione dei dati, che spesso sono espressi in un URL, dove nella *query string* hai la possibilità di vedere queste informazioni.

Tu mi controlli a un certo istante che ho le credenziali, i diritti, ecc., e poi facciamo altre cose. Poi senza più controllare mi fai usare servizi, ti esponi ad attacchi. Nel momento in cui ho un prezzo che il server ha verificato e che hai inserito nel form, io lo modifico, ma tu non controlli, allora ti esponi ad attacchi. Tutto questo ha a che fare con una logica di programmazione.

Lezione 4

Giovedì 26 settembre 2019 – Prof. Mirco Marchetti

Esercitazione: buffer overflow

La funzione *lame* dichiara un array statico di 30 caratteri che si chiama *small*. Dopo di che chiama la *gets* (che legge una stringa da input) e la *puts* (che stampa in output una stringa). La parte vulnerabile sarà la *gets*, perché processa un input. Quando compilo uso il *-32* che vuol dire "compila per sistemi a 32 bit": lo facciamo per convenienza dell'esercitazione. Con *-z execstack* disabilitiamo un controllo fatto dal sistema operativo. Abbiamo visto che vogliamo mettere uno shellcode nello stack e far puntare l'IP sullo shellcode. I sistemi operativi prevedono di evitare di eseguire del codice che è nello stack: noi disabilitiamo questa protezione per poter infilare lo shellcode nello stack, senza dover far puntare l'IP da un'altra parte. Se sforo la dimensione del buffer, il programma termina con errore, perché non può ripristinare lo stato dello stack. *Segmentation fault (core dumped)*, significa che quando un programma è in debug, se c'è un errore, memorizzo lo stato del programma in un *core file*, che posso osservare. Con un debugger ci rendiamo conto che il programma è stato terminato dall'OS con un segnale SIGSEGV. Se osserviamo i registri, vediamo che in EIP (dove si salva l'IP) osserviamo che l'indirizzo non è quello valido. Dato che l'OS si è accorto che il programma ha provato ad uscire dal suo segmento di memoria riservato, allora gli ha mandato un SIGSEGV. La soluzione sarebbe controllare la dimensione della stringa di input.

Un attaccante però non vuole fare solo DoS, ma potrebbe mettere nell'IP un indirizzo valido nel segmento. Sicuramente l'indirizzo della funzione *lame* è valido. Col debugger possiamo disassemblare la funzione main e ottenere il codice assembly, dove troviamo l'indirizzo della funzione main. Però non tutti i caratteri sono stampabili, quindi creiamo un programma *ret.c* per sottomettere degli input a *lame.c*. ret riempie un buffer con l'indirizzo della funzione *lame*. Creato il buffer, poi lo stampo in output, per poi poterlo sottomettere a *lame*. Continuiamo ad avere dei segmentation fault. L'indirizzo anziché essere *0x08049186* è *0x91860804*. C'è un problema di allineamento: modifico *ret.c* shiftando di due byte. Attenzione: devo inizializzare le prime due posizioni nel buffer, perché se c'è uno 0, questo viene identificato come terminatore di stringa, altrimenti *puts* termina. L'effetto è che la funzione viene reiterata.

Vogliamo ora inserire uno shellcode in *vulnerable.c*, che è simile a *lame.c*, ma accetta un parametro che salva in un buffer da 500 byte. Vogliamo inserire lo shellcode preceduto una sled di NOP (*0x90*) e seguito dall'indirizzo di ritorno tante volte. Lo shellcode è un insieme di operazioni già compilate. Per conoscere l'indirizzo uso una direttiva del C per eseguire dell'assembly. Faccio una *move* di *esp*, che è il registro dove salvo il puntatore allo stack, in *eax*, dove si salva il valore di ritorno. Uso questo valore per provare a calcolare il mio indirizzo di ritorno. Per quest'esercitazione, disabiliteremo la randomizzazione degli spazi degli indirizzi, in modo che *vulnerable* ed *exploit* siano contigue in memoria. Quindi posso cercare indirizzi più avanti di *esp*, e con un ciclo, a tentativi, raggiungerò l'IP di *vulnerable*. Riempio il mio buffer di indirizzi di ritorno. Poi la prima metà del buffer la sovrascrivo con delle NOP. Poi mi piazzo a metà del buffer e ci metto lo shellcode. Nota bene: lo shellcode viene eseguito con i privilegi del software vulnerabile.

Lezione 5

Lunedì 30 settembre 2019

Intrusioni (1)

Vogliamo ora capire come sfruttare queste vulnerabilità. I nostri rischi nascono da una combinazione di vulnerabilità e minacce. Parliamo di attacchi mirati. Ci sono due tipi di attacchi: la pesca a strascico (metto una mina e chi prendo prendo) e la pesca mirata (ho un obiettivo specifico). L'attaccante non sta giocando: i giochi hanno regole, qua non ne abbiamo. L'obiettivo poi non è singolo: non mi interessa solo rubare i dati, ma fare tante altre cose. Un caso può essere: sono entrati gli attaccanti, me ne sono reso conto tardi, ma cosa è successo nel frattempo? Ci possono essere danni evidenti o non evidenti: noi non possiamo sapere gli obiettivi degli attaccanti. I danni principali sono il furto dati, soldi, reputazione e danni all'operation.

Come possiamo essere certi che gli attaccanti hanno causato un danno o un furto? Possono esserci dei log di sistema. Vale la regola: *no log, no crime*. Se preventivamente non hai sistemi di monitoraggio, e non ti vengono a ricattare o a chiedere un riscatto, tu impazzisci: non puoi sapere cosa ti hanno rubato, cos'è successo. Se invece hai predisposto strumenti di monitoraggio (log presenti, immodificabili, trasportati all'esterno) allora puoi fare delle stime adeguate del danno.

Abbiamo detto che ci sono diverse vulnerabilità, a livello di rete, di OS, di software dei server, di applicazioni e di persone. Nella stragrande maggioranza dei casi, queste sono le prime vulnerabilità che sfrutto: software non aggiornato, configurazione errata,

configurazione inadeguata alla sicurezza, ecc. I ransomware si possono diffondere tramite file-sharing, tramite risorse condivise. A guardarci meglio, la colpa è sempre dell'essere umano: errori o negligenze che si riflettono sui sistemi. Tuttavia, le vulnerabilità che troviamo sono sempre obiettivi secondari: non mi interessa compromettere un software o un server, ma vedo le vulnerabilità come mezzi per arrivare ai veri obiettivi.

La vulnerabilità numero 1 sono le persone, seguita dal software e dalla rete. Per la rete era vero all'inizio: i primi provvedimenti di sicurezza sono arrivati a livello di rete. Il software fa schifo, perché le persone non l'hanno programmato bene. Le persone dovrebbero essere libere di cliccare dove pare a loro, ma chi attacca sfrutta le loro vulnerabilità e quelle del software per creare danni.

Come si attacca un'azienda? Dobbiamo raccogliere informazioni: chi ci lavora, come funziona l'azienda, se ci sono i turni, ottenere le mappe, ecc. Bisogna informarsi il più possibile a livello tecnologico e di personale. Chi sono i dipendenti, quali sono le loro relazioni, se usano dispositivi; quali sono le infrastrutture, se usano il cloud, qual è la rete coi fornitori e coi clienti. Quindi si fa **intelligence**. Mi informo tramite il sito web, tramite i social (LinkedIn), online. Se l'azienda non è lontana, ci sono attacchi in cui prima di entrare in azienda decido di frequentare prima i luoghi esterni frequentati dai dipendenti, cerco di ottenere informazioni, anche per periodi prolungati.

Devo capire a quale anello agganciarmi: vulnerabilità delle persone, del software, della rete? Mi faccio un dossier su quell'azienda e decido di muovermi a livello tecnologico, psicologico o entrambi. Quindi **scelgo gli obiettivi**. Successivamente **entro in contatto** per capire quale sia l'anello debole, che può essere trovato nelle persone, negli strumenti, nelle infrastrutture, nei servizi informatici e nelle relazioni con fornitori e con clienti. Infine, **sfrutto le vulnerabilità**. Attenzione: il processo non è lineare, ma spesso si torna indietro e si ripetono le varie fasi.

Sapendo ciò, cosa può fare il difensore? L'unica cosa che possiamo fare è **agire a livello preventivo**: sapendo questo, cerco di ridurre lo spazio a cui si possono agganciare coloro che attaccano. Nel contatto, il tuo attaccante si espone, quindi possiamo accorgercene. Rilevo una mail strana e confronto con gli altri. Oppure scansiono le porte della rete. Il contatto è il momento della *early-detection*. Dovremmo avere dei sensori tecnologici e dei sensori umani (difficile).

Un'azienda ha una sua gerarchia, ha diverse competenze, non necessariamente informatiche. Gli attacchi non arrivano soltanto dall'esterno. Un'azienda ha senza dubbio il suo personale, organizzato in maniera più o meno gerarchica. Un'azienda crea valore aggiunto: prende risorse dai propri fornitori (beni, dati, soldi, informazioni, energia, ecc.); grazie ai propri strumenti aggiunge valore e le trasferisce ai propri clienti. Questi possono essere altre aziende oppure consumatori. Un'azienda è sempre immersa in un flusso di beni che vengono acquisiti, trasformati e consegnati. Nessun'azienda è autosufficiente: ci sono tantissime persone che entrano e che escono (manutentori, avvocati, commercialisti, ecc.). Quindi l'azienda non è inespugnabile.

Vulnerabilità umane

Oggi ci sono tantissime informazioni sulle persone in rete. Una volta che mi sono fatto un'idea di chi lavora in un'azienda, il mio target potrebbe essere chiunque: un fornitore, un consulente, un soggetto interno poco controllato. Da queste persone voglio scoprire le vulnerabilità umane, che sono innumerevoli: guadagno, debiti, ricattabilità, curiosità, ignoranza, incompetenza, eccesso di fiducia, sesso, ecc. Le cose si sposano con i dispositivi sempre più legati alle persone: il nostro più grande problema è il BYOD (Bring Your Own Device), ovvero a lavoro si portano spesso i propri dispositivi. La tecnica si chiama **social engineering**, che non ha nulla a che fare con l'ingegneria, ma con la psicologia. Ci sono tantissimi meccanismi per sfruttare queste vulnerabilità. Si sfrutta ad esempio il desiderio di aiutare: mi daresti una mano, mi faresti una copia? Se siamo addestrati, possiamo verificare che tramite alcuni contatti si diventa meno sensibili e si abbassa la guardia. Quindi bisogna essere paranoici.

Il mezzo più diffuso per poter raggiungere tanti obiettivi aziendali è l'uso di mail mandate nel mucchio: **SPAM**. L'obiettivo è far cliccare su una mail, fagli aprire l'allegato o un link e sfruttare delle vulnerabilità. È chiaro che dall'altra parte bisogna stare attenti. Segnalare per fare un'early-detection. Avere un sistema di segnalazione di SPAM più o meno mirato può essere utile. Anche il telefono può essere utilizzato per avere più informazioni. Bisogna stare attenti alla persona con cui parliamo, stare attenti a capire con chi parliamo, se è davvero lui, ecc.

Si è passati allo **spear-phishing**, ovvero a una mail confezionata e mirata verso quell'azienda: stavolta non pesco a strascico. Se siamo noi il target scelto, non è facile proteggersi.

Avversari interni

La maggior parte di attaccanti interni sono spie industriali, dipendenti di aziende partner, consulenti, ex dipendenti, stagisti, persone negligenti, imprudenti. Uno dei motivi più diffusi è la vendetta: perché lui è stato promosso e io no? Anche quando l'azienda è in crisi dovrei aumentare le misure di difesa. Ciascuna azienda dovrebbe osservare il livello di allerta sulla base dello stato delle cose. Così come quando un'azienda sta per essere comprata. La maggior parte dei casi invece è data da azioni colpose: