

Attacchi a Zone Adiacenti di Memoria e Teoria della Complessità delle Vulnerabilità

Angelo P.E. Rosiello

Politecnico di Milano: DEI, Dipartimento di Elettronica ed Informazione, Milano, Italy

Relatore: Prof. Fabrizio Ferrandi, Politecnico di Milano

Abstract—Nel corso degli ultimi anni sono state sviluppate varie tecniche di sfruttamento delle vulnerabilità insite e latenti nel codice di software largamente distribuiti. Lo scopo di queste tecniche è quello di fornire all'attaccante strumenti efficaci ed efficienti per ottenere il pieno controllo della macchina target. Alcuni esempi ormai noti sono gli exploit per stack overflow, heap overflow, format string bug, etc. Una tipologia di attacco meno nota è invece relativa allo sfruttamento di zone adiacenti di memoria nello stack, sovrascrivendo l'ultimo byte terminatore dei buffer. In questo articolo verrà illustrato lo stato dell'arte degli attacchi a zone adiacenti di memoria ed altresì verrà considerata un'estensione del problema non conosciuta in letteratura, conseguenza di accessi erronei alla memoria. Per concludere verrà analizzato e proposto un nuovo argomento, al confine tra l'ingegneria del software e la sicurezza informatica: la complessità delle vulnerabilità. Lo scopo è quello di fornire una metodologia che consenta di comprendere meglio la correlazione tra complessità delle vulnerabilità e la qualità dei processi di sviluppo delle software house.

I. INTRODUZIONE

In quest'era, dominata dalle tecnologie intelligenti distribuite su larga scala, il problema della sicurezza diventa quanto mai attuale e prioritario. In tutti i paesi sviluppati e quelli in via di sviluppo, la maggioranza della popolazione possiede un pc o per lo meno un cellulare, palmare o qualsiasi altro dispositivo su cui venga eseguito del software. Lo scopo del software è di offrire determinate funzionalità all'utente, interagendo con il dispositivo fisico su cui è installato. Nel corso del tempo sono state pensate tecniche e metodologie in grado di consentire ad un agente esterno di ottenere accesso non autorizzato a sistemi o ad informazioni riservate al fine di commettere frodi, estorsioni, spionaggio industriale, furti di identità, attacchi e dinieghi di servizio, etc.

Una prima tassonomia degli attacchi più comuni esistenti può essere stilata in base alla natura dell'attacco stesso:

- Ingegneria Sociale – l'arte di persuadere la vittima al fine di estorcere informazioni sensibili, quali la password di accesso alla sua e-mail, postazione di lavoro, etc., o eseguire programmi malevoli, senza la sua piena consapevolezza o esplicita volontà.
- Exploit-based – sfruttamento di vulnerabilità insite nel codice di un programma al fine di prendere il

controllo o impedire il corretto funzionamento della macchina target per via remota o locale. A questa categoria appartengono tutti quegli attacchi aventi come obiettivo la sovrascrittura dell'istruzione pointer (IP) del processore puntando a delle istruzioni arbitrarie opportunamente iniettate in memoria (e.g. shellcode nello stack). In generale possiamo considerare attacchi exploit-based anche quelli che consentono l'esecuzione di codice malevolo o di azioni contro la volontà dell'utente, pur non impattando direttamente sull'IP, e tuttavia sfruttando bug del software. I buffer-overflow, format-string bugs, signal handler race conditions, XSS, sql-injection, etc., appartengono a questa classe di attacchi.

- Shadow Server/Software – perfetta copia di un server/software, oggetto dell'attacco, in grado di simulare l'interattività con l'utente (vittima) al fine di catturare informazioni sensibili della vittima, come descritto in [5][6]. Questa tipologia di attacco è da considerarsi una via di mezzo fra l'ingegneria sociale e gli attacchi exploit-based, infatti, una sua istanza è ad esempio il *Phishing exploit-based*[7] che sfrutta alcune vulnerabilità dei browser più utilizzati per installare malware (e.g. keyloggers, backdoors, etc.) e carpire le password della vittima o altre informazioni.
- Brute Forcing – tentativo di indovinare/scoprire i dati dell'utente (password, numero di carta di credito, etc.) provando tutte le possibili combinazioni di dominio in maniera deterministica ed algoritmica.

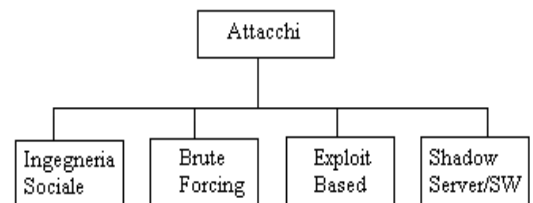


Figure 1. Tassonomia delle principali tipologie di attacco.

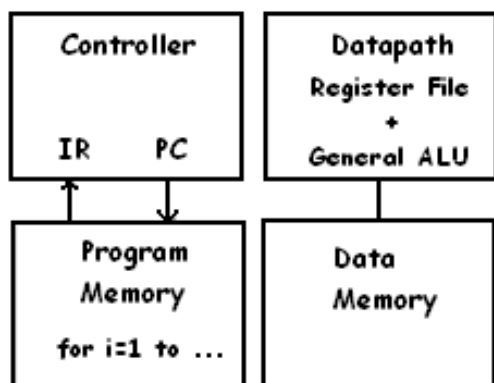


Figure 2. Visione astratta dell'architettura di un processore general-purpose.

In questo articolo l'enfasi sarà posta sugli attacchi exploit-based, sebbene statisticamente siano quelli meno impiegati dai *cracker* quando confrontati alle altre tipologie di attacco precedentemente descritte (e.g. Ingegneria Sociale), poiché richiedono un'elevata competenza dell'attaccante nell'ambito della computer science. In particolar modo nella sezione II ci concentreremo, anche analizzando dei semplici esempi, su una tipologia di attacco poco nota in letteratura, e cioè la concatenazione di zone adiacenti di memoria (i.e. buffers) nello stack. In questo caso, sfruttando opportuni errori di programmazione, l'attaccante riuscirà a fondere due o più aree di memoria e possibilmente a provocare uno stack/heap overflow, a seconda delle politiche interne del processore. Oltre allo stato dell'arte verrà introdotto un nuovo scenario di attacco che consente di produrre gli stessi effetti dei classici attacchi proposti in letteratura, ma questa volta in maniera più indiretta e quindi meno evidente agli occhi dell'analista. Nella sezione III invece, cercheremo di gettare le fondamenta per la teoria della complessità delle vulnerabilità del software. L'idea è di identificare un primo insieme di dimensioni che consentano di "misurare" la qualità della sicurezza del software (ed implicitamente dei processi di sviluppo delle software house) in relazione al livello di severità degli attacchi a cui è soggetto durante l'intero ciclo di vita. Infine, alcune conclusioni nel paragrafo IV termineranno l'articolo.

II. ATTACCHI A ZONE ADIACENTI DI MEMORIA

Prima di affrontare gli attacchi alle zone adiacenti di memoria, è opportuno descrivere brevemente, ad un alto livello di astrazione, l'organizzazione architeturale dei processori general-purpose più comuni, ed in maniera più dettagliata l'organizzazione della memoria. Principalmente si farà riferimento all'architettura Intel[1], poiché risulta essere una delle più diffuse e conosciute. Le considerazioni tratte potranno comunque essere estese, ovviamente sotto opportune condizioni, ad altre architetture comuni, quali SPARC[2], PowerPC[3], etc., e a classi di processori ASIP (i.e. Application Specific Instruction-Set Processors) e single-purpose processors.

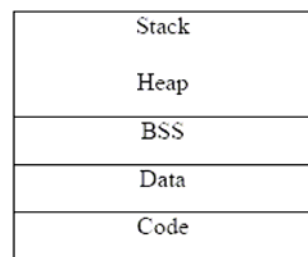


Figure 3. Organizzazione della memoria per un processo.

A. Architettura di un processore general purpose

Un processore general-purpose è un *device* in grado di leggere istruzioni dalla memoria a tempo di esecuzione, ed eseguirle nel modo più efficiente possibile. L'obiettivo del progettista di questa categoria di processori è di realizzare un device in grado di eseguire funzioni ed applicazioni di varia natura, che non sono note a-priori. Il processore consiste di tre componenti principali: a) il controllore; b) il datapath; 3) la memoria (dati e di programma, a seconda dell'architettura di riferimento, i.e. Harvard o Princeton). Una caratteristica di questa tipologia di processori è il datapath, poiché deve consentire l'esecuzione di istruzioni generiche. A questo scopo il datapath si compone di una o più *general-purpose arithmetic logic units* (ALUs) e di un insieme abbastanza grande di registri (register file). Le funzionalità del sistema sono insite nel software che risiede nella memoria di programma. Il controllore si occupa di gestire, fra le altre cose, la lettura delle istruzioni, impostando opportunamente il program counter (o instruction pointer), e caricando l'istruzione corrente nell'instruction register. Il datapath si occupa dell'elaborazione e della memorizzazione dei risultati in memoria dati. In Figure 2. è possibile osservare la generica architettura astratta di un processore general-purpose.

B. Organizzazione della memoria

Quando il codice oggetto di un programma viene letto e caricato in memoria per l'esecuzione, esso prende il nome di *processo*. Il sistema operativo si occupa di caricare le istruzioni e di allocare differenti zone di memoria dati per gestirne la corretta esecuzione. L'intero spazio di memoria allocato per un processo prende il nome di *spazio di indirizzamento* e consiste di cinque settori principali:

- **Segmento Codice:** questa sezione contiene il codice eseguibile del programma, ovvero le istruzioni presenti nel codice oggetto.
- **Segmento Dati e BSS:** entrambi i settori sono dedicati a memorizzare le variabili globali e vengono allocati a tempo di compilazione. Il settore BSS contiene variabili non inizializzate che possono assumere valori concreti run-time, mentre il segmento dati è riservato per dati statici.
- **Stack:** le variabili locali automatiche vengono allocate in questa sezione di memoria che è anche particolarmente utile per il passaggio di parametri fra funzioni, e per la memorizzazione delle variabili di contesto. Lo stack cresce verso il basso considerando le politiche Intel [1].

```
//Esempio 1
int main(){
char buffer1[]="ab";
char buffer2[]="cd";
...;
return 0;

//Stack
[c]
[d]
[0x0] (X)
[a]
[b]
[0x0]
```

Figure 4. Esempio 1.

- Heap: segmento che rappresenta di fatto tutta la restante parte di memoria di un processo. Lo heap cresce in direzione opposta dello stack e lo spazio viene allocato dinamicamente.

Nella sezione II.C verrà illustrato lo stato dell'arte degli attacchi alle zone adiacenti di memoria, con particolare riferimento allo stack. Tuttavia è possibile applicare i medesimi concetti anche ad altre sezioni di memoria, come ad esempio lo heap, laddove le politiche di gestione del processore lo consentano.

C. Attacchi a zone adiacenti di memoria: stato dell'arte

Negli ultimi anni sono stati rilasciati alcuni articoli[4] in merito allo sfruttamento delle zone adiacenti di memoria nello stack, con conseguente produzione di uno stack-overflow. Il problema insorge nel momento in cui l'ultimo carattere terminatore di una stringa (null-byte) viene sovrascritto, ed una altra stringa la precede in memoria.

Infatti, quando un buffer viene dichiarato, esso viene terminato in memoria con un carattere standard (e.g. '\0') per separarlo dalle restanti sezioni allocate nello stack. In Figure 3. è possibile osservare come appare lo stack run-time quando il programma viene eseguito. Se l'attaccante riuscisse in qualche modo a provocare la sovrascrittura del carattere terminatore marcato con (X) in Figure 3. , allora buffer1 e buffer2 risulterebbero concatenati, così che puntando buffer2 verrebbe restituita l'intera stringa "cdXab", invece di "cd".

Uno degli aspetti chiave per l'attaccante che volesse sfruttare questa tipologia di attacco consiste nel trovare funzioni largamente usate ed in qualche modo error-prone. Una nota funzione standard che si presta a far generare concatenazioni di stringhe nello stack è la seguente:

```
char *strncpy(char *dst, const char *src, size_t len) (1)
```

Come descritto nell'opportuna guida di riferimento[8], la funzione (1) copia un numero di caratteri pari al più a *len* da *src* a *dst*. Se il buffer *src* presenta un numero di caratteri inferiore a *len* allora la restante parte di *dst* è riempita con caratteri terminatori (i.e. '\0') altrimenti *dst* non viene terminato. Nell'esempio in Figure 4. se l'utente desse in input una stringa avente cinque o più caratteri, ad es. "iceburn", *buffer2* verrebbe concatenato a *buffer1*, e puntando *buffer1* si otterrebbe in output la stringa "icebiceburn".

La semplice concatenazione di stringhe nello stack finora descritta dà origine ad errori funzionali ma non rappresenta

```
//Esempio 2
int main{
char buf1[8];
char buf2[4];
fgets(buf1, 8, stdin);
strncpy(buf2, buf1, 4);
function(buf2);}

void function( char buf2[32] ) {
char buf3[8];
strcpy( buf3, buf2 );
}
```

Figure 5. Esempio 2.

ancora una minaccia a livello di sicurezza. Al fine di prendere il controllo della macchina target, l'attaccante deve produrre in qualche modo uno stack/heap overflow e ciò potrebbe accadere nel momento in cui venisse copiato un buffer, soggetto a concatenazione, in un altro buffer, ignorando gli effetti della concatenazione. In Figure 5. quando viene invocata 'function()', *buf2* è copiato in *buf3* che ha una dimensione (8) strettamente maggiore di *buf2* (4). Tuttavia post-concatenazione *buf2* restituirà potenzialmente una stringa di lunghezza pari alla somma dei caratteri contenuti in *buf1* e *buf2*, ovvero 12. Questo può provocare un classico stack-overflow con conseguente sovrascrittura dell'IP del processore.

In teoria la tipologia di attacco finora descritta considerando lo stack come ambiente di memoria, può essere riportata anche allo heap, ma tipicamente le zone di memoria nello heap non vengono allocate in *adjacent fashion* come nello stack, quindi non è possibile sapere a-priori se "colpire" il carattere terminatore di un buffer provocherà la concatenazione con un altro buffer in memoria.

D. Attacchi a zone adiacenti di memoria: un nuovo scenario

Nella sezione II.C è stato descritto lo stato dell'arte degli attacchi alle zone adiacenti di memoria che si basano fondamentalmente sull'uso inappropriato di alcune funzioni standard, quali ad esempio *strncpy()* o *strncat()*. Tuttavia esiste un ulteriore scenario di attacco che dà all'attaccante la possibilità di concatenare due regioni di memoria.

Per capire in maniera diretta e semplice il problema consideriamo l'Esempio 3 in Figure 6. Quando la routine *recv()* viene invocata ed eseguita correttamente, essa ritorna il numero di byte ricevuti. La variabile 'i' tiene traccia del numero di byte ricevuti e viene utilizzata come indice del *buffer1*. In condizioni "ideali" il programma funzionerebbe correttamente, ma per la *Legge di Murphy* non possiamo ignorare il fatto che qualcosa potrebbe andare storto, e quindi ci aspettiamo un errore dall'esecuzione della *recv()*. In caso di errore, *recv()* restituisce il valore intero '-1' ed accedendo, mediante l'indice 'i', in 'buffer1[-1]', il byte terminatore di *buffer2* verrà sovrascritto, provocando la concatenazione di *buffer1* e *buffer2*.

Questa tipologia di vulnerabilità è un problema complesso sia per l'attaccante che per l'analista. Infatti, accedendo alla memoria run-time non è possibile per l'analista considerare tutti i cammini possibili del *DFG* (Data Flow Graph)[9]. Altresì, se

```

//Esempio 3
int main() {
    int i=0;
    char buffer1[64];
    char buffer2[64];
    /* some code here that fills buffer1 and buffer2 */
    i=recv(...);
    //controllo dell'overflow
    if(i>63) i=63;
    buffer1[i]=i;
    .....;
    return 0;
}

```

Figure 6. Esempio 3.

ad esempio la variabile 'i' assumesse valori di ritorno da più funzioni annidate fra loro ed invocanti librerie diverse, è necessario avere piena consapevolezza dell'intero flusso dati. Dal punto di vista dell'attaccante è necessario creare le condizioni perché avvenga il concatenamento dei buffer e ciò può essere tutt'altro che banale e richiedere più fasi di attacco.

III. TEORIA DELLA COMPLESSITÀ DELLE VULNERABILITÀ

A seguito della diffusione degli attacchi in rete, molti studi sono stati condotti in merito alla modellazione dei rischi, minacce, valutazione dei danni, etc., relativi alle vulnerabilità ed ai tipi di attacco più comuni. Uno dei principali metodi adottati attualmente per "pesare" le vulnerabilità è di associare un livello di severità all'*advisory*. Molte software house, sebbene adottino processi di sviluppo consolidati e standardizzati, accompagnati da tecniche di testing/analisi avanzate, sono spesso soggette a vulnerabilità ad alta severità. Ciò ci fa intuire che evidentemente è assente una qualche metrica che ci consenta di pesare le vulnerabilità individuate.

Uno dei principali contributi di questo articolo è di approfondire la complessità delle vulnerabilità cercando di fornire alcuni criteri che consentano di saggiare la maturità del software dal punto di vista della sicurezza, in risposta alle vulnerabilità scoperte durante il suo ciclo di vita. In questo modo ci auguriamo anche di poter stabilire se un'azienda può essere considerata matura nello sviluppo di software sufficientemente sicuro, oltre che affidabile, leggibile, manutenibile, etc. [9].

A. Alcune dimensioni per la complessità delle vulnerabilità

Per misurare la complessità di una vulnerabilità sono state prese in considerazione sei dimensioni emblematiche che potranno essere estese in seguito:

1. Novità vulnerabilità/attacco: quanto è nuova alla comunità la tipologia di attacco o vulnerabilità identificata?

Dal punto di vista della vulnerabilità, l'uso insicuro della funzione '*strcpy()*' ad esempio, è ormai largamente conosciuto dalla comunità, così come dal punto di vista dell'attacco, le tecniche di sfruttamento degli stack-overflow[10] sono largamente note alla comunità.

2. Immediatezza dell'attacco: perché l'attacco abbia successo è necessario che siano percorse molte fasi di esecuzione del software e che siano applicate differenti tecniche di sfruttamento (ingegneria sociale, exploit-based, etc.) o la riuscita è pressoché immediata?
3. Complessità del codice: quanto è latente la vulnerabilità nel codice? Alcune metriche che è possibile adottare sono:
 - a. Immediatezza nell'individuazione: semplicità/complessità nell'individuazione della vulnerabilità utilizzando strumenti e metodi attuali e riconosciuti (e.g. ispezione manuale, strumenti testing automatico, etc.).
 - b. Profondità del codice: quanto in fondo al codice bisogna andare per raggiungere la vulnerabilità, in relazione alla profondità complessiva del software in esame. In questo caso è necessario tenere presente anche eventuali pre-condizioni da validare.
 - c. Livello di in direzione: se la vulnerabilità è legata squisitamente al codice del software in analisi, o si appoggia ad API di terze parti, funzioni esterne, etc.
 - d. Metodologie tradizionali di analisi del codice, come ad esempio:
 - i. Complessità di Halstead [12]: misura della complessità di un modulo, con enfasi sulla complessità computazionale, sulla base di operatori (i.e., elementi che rappresentano azioni sui dati) operandi (i.e., elementi che rappresentano i dati).
 - ii. Function Point [13]: cercano di quantificare le funzionalità offerte dal sistema.
 - iii. Linee di codice: numero delle linee di codice del modulo/pacchetto considerato.
 - iv. McCabe Complexity [11]: numero di percorsi linearmente indipendenti in un modulo. Il calcolo è fatto sul CFG del codice.
4. Complessità dell'attacco: numero di input da manipolare, complessità delle manipolazioni, numero di interfacce a cui accedere e controllare per apportare l'attacco.
5. Ubiquità: numero di versioni, configurazioni, piattaforme affette dalla vulnerabilità.
6. Potere dell'attaccante: grado di controllo richiesto all'attaccante sull'ambiente in cui l'attacco ha luogo. E.g.: diritti di accesso, numero di permessi, controllo su risorse esterne, etc.

B. Confronti di complessità

Dopo aver definito le dimensioni che caratterizzano la complessità delle vulnerabilità, cerchiamo di confrontare la complessità degli attacchi alle zone adiacenti di memoria con una nota tipologia di vulnerabilità, cioè i buffer-overflow. Per essere chiari e schematici compiliamo una tabella in cui compariranno le dimensioni individuate in III.A.

Dimensioni	Buffer Overflow	Zone Adiacenti
Novità	Molto Conosciuto	Poco Conosciuto
Immediatezza	Immediato	Poco Immediato
Complessità Codice	Dipende dall'applicazione	Dipende dall'applicazione
Complessità Attacco	Medio/Bassa Complessità	Medio/Alta Complessità
Ubiquità	Dipende dall'applicazione	Dipende dall'applicazione
Potere Attaccante	Basso	Medio/Alto

TABLE I. CONFRONTO COMPLESSITÀ TRA I BUFFER OVERFLOW E LE ZONE ADIACENTI DI MEMORIA

Come è possibile notare dalla TABLE I. il livello di complessità che caratterizza una vulnerabilità relativa alle zone adiacenti di memoria è per lo meno strettamente maggiore di quello dei buffer overflow. Fra le dimensioni analizzate, la complessità del codice e l'ubiquità sono fortemente dipendenti dall'applicazione soggetta alla vulnerabilità, e non è possibile quindi fornirne una valutazione *a-priori*.

IV. CONCLUSIONI E SVILUPPI FUTURI

Durante l'analisi ed il testing del software è necessario considerare entrambi gli scenari descritti in questo articolo causanti attacchi a zone adiacenti di memoria, e nella fattispecie, l'uso scorretto di alcune funzioni di libreria ed accessi particolarmente "indiretti" alla memoria.

Grazie alla teoria della complessità delle vulnerabilità ci auguriamo di aver gettato le fondamenta per una nuova metodologia atta a stabilire la qualità del software, ed implicitamente dei processi di sviluppo delle software house, avendo come obiettivo chiave la sicurezza dei prodotti finali.

Lavori futuri riguardano l'esplorazione e l'allargamento delle dimensioni individuate per valutare la complessità delle vulnerabilità, ma altresì lo studio di metodologie complete e robuste che consentano il raffronto delle classi di vulnerabilità esistenti.

ACKNOWLEDGMENT

Vivi ringraziamenti a Steve Christey, della "Common Vulnerabilities and Exposures", per il significativo contributo in merito alla complessità delle vulnerabilità.

REFERENCES

- [1] Intel®. "IA-32 Intel® Architecture Software Developer's Manuals." - http://www.intel.com/design/pentium4/manuals/index_new.html
- [2] The SPARC Architecture Manual, Version 8 - <http://www.sparc.org/resource.htm#V8>
- [3] PowerPC Microprocessor Family: The Programmer's Reference Guide - <http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600741775>
- [4] Twitch. "Exploiting Non-adjacent Memory Spaces." - Phrack Magazine n° 56, 2000.
- [5] A.Lioy. Volume V Sistemi Distribuiti - Franco Angeli, 2001.
- [6] Angelo Rosiello. "Shadow software Attack still works." - <http://www.rosiello.org>, April 2004.
- [7] Engin Kirda, Christopher Kruegel. "Protecting Users Against Phishing Attacks." - The Computer Journal Vol. 00 No. 0, 2005.
- [8] Linux Man <http://man.linuxquestions.org/index.php?query=stncpy§ion=3&type=2>
- [9] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli. "Fundamentals of Software engineering." - 2nd edition Prentice Hall, 2003.
- [10] Angelo P.E. Rosiello. "La goccia che fa traboccare il vaso." - HackerJournal n° 32 (pag. 29-31), 2003.
- [11] McCabe, Thomas J. "A Complexity Measure." - IEEE Transactions on Software Engineering, SE-2 No. 4, pp. 308-320, December 1976.
- [12] M. H. Halstead. Elements of Software Science, Operating, and Programming Systems Series, y, 1977.
- [13] Behrens, Charles A. "Measuring the Productivity of Computer Systems Development Activities with Function Points" - IEEE Transactions on Software Engineering, Vol. SE-9, no. 6, November 1989.
- [14] William Stallings. "Operating Systems: Internals and Design Principles" - 5th ed., Prentice-Hall, 2005.