

Il superbug di Windows (MS04-007)

a cura di **Massimiliano Domenico Dessi, Giovanni Magliocca**

Overview

- **Introduzione**
- Descrizioni tecniche
- Buffer overflowed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

La più critica falla di Windows

- **12 febbraio 2004:** Microsoft corregge alcune vulnerabilità presenti in Windows e nel servizio WINS (Windows Internet Naming Service)
 - Interessano tutte le versioni di Windows basate su kernel NT – NT4, 2000, XP, 2003
 - Possono consentire ad un cracker di ottenere il controllo di un sistema da remoto

La più critica falla di Windows

- Le falle sono di tipo **buffer overflow**
 - Contenute in una libreria che implementa l'ASN.1
- **ASN.1 (Abstract Syntax Notation 1)**
 - Componente di Windows dalla versione NT4
 - Consente ai programmi di interpretare correttamente i dati ricevuti da dispositivi o altri programmi
 - Molto usata dai sottosistemi di sicurezza di Windows, quindi sono molti i modi per sfruttare la vulnerabilità

La più critica falla di Windows

*"Un aggressore che sfrutti con successo questa vulnerabilità potrebbe eseguire del codice con i privilegi di system", si legge nell'avviso di sicurezza **MS04-007** di Microsoft.
E ancora: "L'aggressore potrebbe poi intraprendere qualsiasi azione, incluso installare programmi, creare nuovi account con i massimi privilegi o visualizzare, modificare e cancellare dati"*

- Tali vulnerabilità sono state classificate da Microsoft con il massimo livello di criticità
 - livello *critical*
 - da qui l'indicazione di tali vulnerabilità con il termine **superbug**

Qualche data...

- **25 luglio 2003:** la eEye Digital Security annuncia in un comunicato che la libreria MSASN1 di Windows è piena di integer overflow
- **25 settembre 2003:** ancora la eEye in un altro comunicato annuncia la scoperta di una seconda vulnerabilità

Sistemi e software affetto

- Sistemi affetti:
 - Microsoft Windows NT 4.0 (tutte le versioni)
 - Microsoft Windows 2000 (SP3 e versioni precedenti)
 - Microsoft Windows XP (tutte le versioni)
 - Microsoft Windows Server 2003
- Software affetto:
 - Microsoft Internet Explorer
 - Microsoft Outlook
 - Microsoft Outlook Express
 - Altre applicazioni che fanno uso di certificati digitali
 - Servizi come Kerberos, Microsoft IIS, NTLMv2 authentication

Overview

- Introduzione
- Descrizioni tecniche
 - Comunicato eEye del 25 luglio 2003
 - Comunicato eEye del 25 settembre 2003
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

eEye: 25 luglio 2003

- La libreria MSASN1 è piena di integer overflow
- La vulnerabilità colpisce qualsiasi client di MSASN1.DLL
 - In particolare LSASS.EXE e CRYPT32.DLL
- Mostriamo alcuni errori aritmetici nella decodifica BER di ASN.1

... ma prima qualche notizia su BER

Schema di codifica BER

- Ogni blocco di dati è codificato come un valore tipato costituito da
 - **tag number**, che descrive come interpretare i valori successivi
 - **lunghezza dei dati**
 - **dati**

... il campo **lunghezza** è l'oggetto della nostra trattazione

Come sfruttare la vulnerabilità

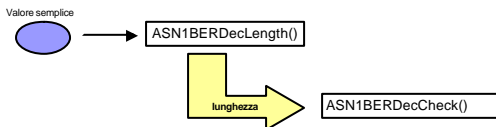
- Fornendo un valore molto grande nel campo **lunghezza** è possibile provocare un integer overflow in una routine di allocazione di memoria

Come sfruttare la vulnerabilità

1. Decodifica del blocco BER
2. Verifica della presenza dei dati
3. Allocazione della memoria
4. Copia dei dati

Come sfruttare la vulnerabilità

1. Quando un valore semplice è decodificato da MSASN1, la funzione `ASN1BERDecLength()` è invocata per recuperare la lunghezza. Tale valore è poi inviato alla funzione `ASN1BERDecCheck()` per controllare che i dati esistano realmente



Come sfruttare la vulnerabilità

1. Decodifica del blocco BER
2. Verifica della presenza dei dati
3. Allocazione della memoria
4. Copia dei dati

Come sfruttare la vulnerabilità

2. `ASN1BERDecCheck()` verifica che

$$[(\text{puntatore_inizio_dati} + \text{lunghezza}) = (\text{puntatore_inizio_blocco_BER} + \text{grandezza_blocco_BER})]$$

Se ciò non avviene, la decodifica viene interrotta e la funzione ritorna con un errore

Come sfruttare la vulnerabilità

1. Decodifica del blocco BER
2. Verifica della presenza dei dati
3. Allocazione della memoria
4. Copia dei dati

Come sfruttare la vulnerabilità

3. Se si tenta di allocare memoria ai dati, la lunghezza decodificata viene passata alla funzione `DecMemAlloc()`.

`DecMemAlloc()` arrotonda la lunghezza a un multiplo di DWORD (supposta uguale a 4 byte) e tenta di allocare il risultato, ovvero effettua la seguente operazione:

`LocalAlloc(LMEM_ZEROINIT, (lunghezza + 3) & ~3)`

Come sfruttare la vulnerabilità

- `LocalAlloc(LMEM_ZEROINIT, (lunghezza + 3) & ~3)`
 - `LocalAlloc`: funzione che si occupa di allocare memoria dallo heap
 - `LMEM_ZEROINIT`: flag che indica che lo spazio allocato sarà inizializzato a 0
 - `(lunghezza + 3) & ~3`: dimensione (multipla di DWORD) del buffer che verrà allocato
 - `DWORD`: abbreviazione di *double word* indica il doppio della dimensione di una word

Come sfruttare la vulnerabilità

1. Decodifica del blocco BER
2. Verifica della presenza dei dati
3. Allocazione della memoria
 - Alcuni esempi...
4. Copia dei dati

Alcuni esempi...

- Supponiamo che la funzione ASN1BERDecLength restituisca un valore di $length = 17$. Analizziamo le operazioni che vengono effettuate:

Termini	Valori in decimale	Valori in binario (32 bit)
length	17	0000000000000000000000000000000010001
3	3	00000000000000000000000000000000000011
length + 3	20	00000000000000000000000000000000000010100
-3	-	11111111111111111111111111111111111100
(length + 3) & -3	12	00000000000000000000000000000000000010100

- Verrà quindi allocato un blocco di memoria di dimensione 20 byte, ovvero $5 * DWORD$

Alcuni esempi...

- Supponiamo che la funzione ASN1BERDecLength restituisca un valore di $length = 4294967294$. Analizziamo le operazioni che vengono effettuate:

Termini	Valori in decimale	Valori in binario (32 bit)
length	4294967294	1111111111111111111111111111111111110
3	3	00000000000000000000000000000000000011
length + 3	4294967297	00000000000000000000000000000000000001
-3	-	11111111111111111111111111111111111100
(length + 3) & -3	0	00000000000000000000000000000000000000

- Verrà quindi allocato un blocco di memoria di dimensione 0 byte, quando in realtà i dati effettivi occupano 4294967294 byte.

Come sfruttare la vulnerabilità

1. Decodifica del blocco BER
2. Verifica della presenza dei dati
3. Allocazione della memoria
4. Copia dei dati

Come sfruttare la vulnerabilità

4. Se *DecMemAlloc()* ha successo
 - Si effettua una *memcpy()* dei dati all'interno del buffer allocato
 - Si usa l'originale lunghezza decodificata (NON il valore arrotondato) come conteggio dei byte di dati

Come sfruttare la vulnerabilità

- Se viene usata una lunghezza nell'intervallo da **0xFFFFFFFF** a **0xFFFFFFFF**
 - *DecMemAlloc()* arrotonda il valore a 0
 - questo avviene in quanto, sommando 3 ai valori nell'intervallo considerato, otteniamo 3 valori non rappresentabili con 32 bit
 - Alloca con successo un blocco di dimensione 0
 - L'indirizzo del blocco allocato è restituito alla funzione chiamante
 - Alla funzione *memcpy()* è passato il valore originale della lunghezza (quello grande, non arrotondato)

Come sfruttare la vulnerabilità

RISULTATO

Completa **sovrascrittura** dell'heap
con cancellazione di **tutta**
la memoria **contigua**
al blocco di dimensione 0

Overview

- Introduzione
- **Descrizioni tecniche**
 - Comunicato eEye del 25 luglio 2003
 - **Comunicato eEye del 25 settembre 2003**
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

eEye: 25 settembre 2003

- Altri integer overflow rendono i software che usano MSASN1 vulnerabili ad una completa sovrascrittura di una porzione di memoria
- Stessa corruzione dell'heap descritta nel comunicato precedente

Schema di codifica di MSASN1

- Nel caso di una stringa di bit
 - Il primo byte di dati è il numero di bit (da 0 a 7) da escludere dalla fine della stringa
 - I byte rimanenti contengono gli
 $(8 * (\text{lunghezza_valore} - 1) - \text{numero_bit_inutilizzati})$
che costituiscono la stringa di bit

Come sfruttare la vulnerabilità

- Si verifica un integer overflow quando si fornisce una stringa di un byte
 - Ovvero, solo il campo **numero_bit_inutilizzati** contenente un valore diverso da zero

Come sfruttare la vulnerabilità

- La funzione *ASN1BERDecBitString()*
 - Gestisce stringhe di bit composte
 - Concatena le stringhe semplici contenute in quella composta

... mostriamo come generare un integer overflow

Come sfruttare la vulnerabilità

- Forniamo una stringa di un byte e con 7 bit inutilizzati
- *ASN1BERDecBitString()* calcola

$$(8 * (1 - 1) - 7) = -7 \text{ bit}$$

lunghezza_valore numero_bit_inutilizzati

Come sfruttare la vulnerabilità

- Al momento della creazione della stringa composta

bit accumulati	(0) +
lunghezza stringa concatenata	(-7) +
bit per arrotondare	(7) =
<hr/>	
0	

Come sfruttare la vulnerabilità

- Il risultato così calcolato è passato alla funzione *DecMemAlloc()*
- Viene allocato un blocco di dimensione zero
- La funzione *ASNbitcpy()* effettua una *memcpy()* usando le lunghezze originarie

Come sfruttare la vulnerabilità

RISULTATO

Completa **sovrascrittura** dell'heap con cancellazione di **tutta** la memoria **contigua** al blocco di dimensione 0

Conseguenze

- Le falle precedentemente descritte possono essere soggette a vari tipi di exploit
 - Possono consentire l'accesso ad un sistema da remoto
 - Possono far guadagnare i privilegi di *system* ad un cracker
 - Possono consentire l'esecuzione di codice potenzialmente dannoso su di un sistema vulnerabile

Overview

- Introduzione
- Descrizioni tecniche
- [Buffer overflow ed exploit](#)
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

Cos'è un exploit

- Exploit
 - Programma (scritto tipicamente in C)
 - Sfrutta le falle di altri programmi
 - Acquisisce **illegalmente** il controllo di un programma vulnerabile
 - Rende possibile l'esecuzione di codice dannoso che "normalmente" non potrebbe essere eseguito
- Molti exploit sfruttano vulnerabilità di buffer overflow

Buffer overflow

- Cosa si intende per buffer?
 - Blocco di memoria contiguo
 - In C un buffer è chiamato *array*
 - Possono essere statici o dinamici
- L'overflow consiste nel riempire un buffer oltre il limite
 - Per realizzare un exploit di buffer overflow vengono sfruttati gli array dinamici

Buffer overflow

- Funzionamento dei buffer sullo stack
- Come generare buffer overflow
 - È necessario capire il sistema con cui il computer gestisce la memoria di un processo.

Memory layout di un processo

- Graficamente, lo spazio di indirizzamento di un processo appare come segue



User Space

- Può essere ulteriormente diviso nelle seguenti sezioni:
 - **codice**
 - Contiene il codice del programma
 - *sola lettura*
 - **stack**
 - Usato per allocare variabili, passare parametri e restituire valori
 - *lettura, scrittura, esecuzione*
 - **dati**
 - Contiene le variabili statiche
 - **dati inizializzati:** *lettura, esecuzione*
 - **dati non inizializzati:** *lettura, scrittura, esecuzione*

Cos'è lo stack

- Uno **stack** è un tipo di dati astratto
 - Proprietà *LIFO* (Last In First Out)
 - Operazioni: *push* e *pop*
 - Elemento *top* su cui si effettuano le operazioni
- Come viene usato uno stack durante l'esecuzione di un generico processo?

Cos'è lo stack

- Lo stack consiste di un insieme di segmenti logici, detti **stack frame**
- Uno stack frame
 - È impilato (pushed) all'invocazione di una funzione
 - È spilato (popped) quando la funzione ritorna
 - Contiene
 - Parametri della funzione
 - Variabili locali alla funzione
 - Dati necessari al ripristino dello stack frame precedente
 - Indirizzo istruzione successiva alla chiamata a funzione

Cos'è lo stack

- È un blocco contiguo di memoria contenente dati
 - Registro **stack pointer** punta al top
 - Registro **frame pointer** (o **base pointer**) punta ad una locazione fissa nello stack frame
 - La base dello stack è un indirizzo fisso
 - La dimensione è variata dinamicamente dal kernel
 - Le operazioni di push e pop sono implementate dalla CPU

Stack frame di una funzione C

- Supponiamo di avere la seguente chiamata a funzione durante l'esecuzione di un processo

```
int f(char *s)
{
    char c[16];
    int i;

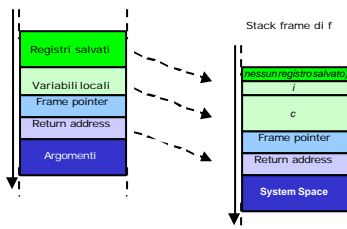
    i = 1;
    ...
    return i;
}
```

Stack frame di una funzione C

- In seguito all'invocazione della funzione *f* vengono eseguite le seguenti operazioni:
 - **push FP**
 - Salva sullo stack il Frame Pointer precedente per poterlo ripristinare al ritorno dalla funzione
 - **mov FP, SP**
 - Copia lo stackpointer SP sul frame pointer FP per creare il nuovo frame pointer
 - **incrementa SP**
 - Al fine di puntare allo spazio riservato alla successiva variabile locale

Stack frame di una funzione C

- Graficamente lo stack frame della funzione appare come segue



Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflowed exploit
- Vulnerabilità di buffer overflowed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

Vulnerabilità di buffer overflow

- Il motivo di fondo di un attacco di buffer overflow è prendere il controllo del programma in esecuzione e finanche dell'host stesso
- Solitamente l'attaccante sceglie come "vittima" un programma che gira come root al fine di avere i massimi privilegi sul sistema attaccato

Vulnerabilità di buffer overflow

- Le vulnerabilità di buffer overflow più conosciute sono:
 - *classico*
 - *frame pointer*
 - *heap*

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
 - Buffer overflow classico
 - Frame pointer overflow
 - Heap-based overflow
 - Integer overflow
- ASN.1 vs BER
- Exploit per MS04-007

Buffer overflow classico

- Un esempio di tale vulnerabilità è dato di seguito

```
int f(char *s)
{
    char c[16];
    int i;

    i = 1;
    strcpy (c, s);
    return i;
}
```

Buffer overflow classico

- Nel codice esempio precedente la funzione *strcpy* copia i dati da *s* a *c* senza effettuare alcun controllo sulla dimensione del buffer di destinazione

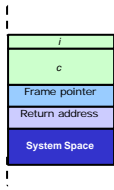
**E' PROPRIO QUESTA
LA VULNERABILITA'!!!**

Buffer overflow classico

- Se la stringa passata in input dovesse avere dimensione superiore a 16 byte, si verificherebbe il tanto paventato buffer overflow in quanto il buffer di destinazione non è in grado di contenere tutti i dati

Buffer overflow classico

- Inizialmente lo stack frame della funzione f appare come segue



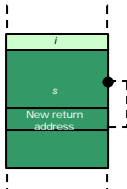
Buffer overflow classico

- In seguito all'invocazione della funzione, i dati in s andranno a sovrascrivere la zona di memoria allocata a c e, se s è abbastanza lunga, anche **frame pointer** e **return address**



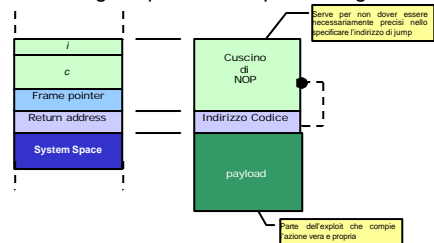
Buffer overflow classico

- La stringa s , passata come argomento, può essere costruita in modo da:
 - contenere delle istruzioni che saranno fatte eseguire sull'host su cui gira il programma
 - andare a sovrascrivere il **return address** affinché punti all'interno dei dati iniettati



Buffer overflow classico

- Usando termini più propriamente relativi all'ambito degli exploit, si ha quanto segue



Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
 - Buffer overflow classico
 - Frame pointer overflow
 - Heap-based overflow
 - Integer overflow
- ASN.1 vs BER
- Exploit per MS04-007

Frame pointer overflow

- Vulnerabilità dovuta a differenti implementazioni di una stessa funzione su piattaforme differenti
- Consente di sovrascrivere solo il primo byte del **frame pointer**
 - per questo è detta *off-by-one*
- Solitamente causata da errori nel controllo della dimensione di un buffer
 - un '=' usato laddove era necessario un '<'

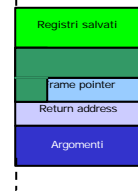
Frame pointer overflow

- Inizialmente lo stack frame di una funzione appare come segue



Frame pointer overflow

- In seguito ad un frame pointer overflow, si riesce a sovrascrivere il primo byte del **frame pointer**



Frame pointer overflow

- Supponiamo di avere il seguente frammento di codice

```
g()
{
    ...
    f();
    ...
}
```

- E supponiamo che la funzione *f* sia affetta da vulnerabilità di tipo frame pointer overflow

Frame pointer overflow

- Al momento dell'invocazione di *f*
 - il vecchio **frame pointer** viene salvato
 - lo **stack pointer SP** è copiato sul **frame pointer FP**

All'uscita della funzione *f*

- il **frame pointer FP** è copiato sullo **stack pointer SP**

**E' PROPRIO QUESTA
LA VULNERABILITA'!!!**

Frame pointer overflow

- Se il **frame pointer FP** è stato modificato a causa della vulnerabilità, risulterà modificato anche lo **stack pointer SP**

■ Conseguenza

- la funzione *g* potrebbe essere usata come trampolino di lancio per far eseguire il codice dell'exploit che potrebbe essere già presente all'interno del programma

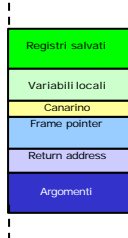
Frame pointer overflow

■ Contromisure

- **stack non eseguibile**
 - ovvero, al segmento di stack (o stack frame) vengono rimossi i diritti di esecuzione
- **canarino**
 - si modifica il compilatore in modo da inserire un ulteriore valore tra le **variabili locali** in entrata ad ogni funzione e il **frame pointer**.
 - all'uscita da ogni funzione si controlla se il canarino (che può essere un valore casuale o semplicemente 0) è ancora "vivo".
 - se non è "vivo", ovvero se non è intatto, significa che è stato sovrascritto (probabilmente a causa di un frame pointer overflow) e l'esecuzione viene interrotta.

Frame pointer overflow

- Usando la tecnica del canarino, lo stack frame di una funzione è modificato come segue



Overview

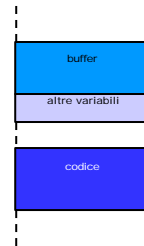
- Introduzione
- Descrizioni tecniche
- Buffer overflowed exploit
- Vulnerabilità di buffer overflowed attacchi
 - Buffer overflow classico
 - Frame pointer overflow
 - Heap-based overflow
 - Integer overflow
- ASN.1 vs BER
- Exploit per MS04-007

Heap-based overflow

- Finora abbiamo visto attacchi mossi a buffer presenti all'interno della regione dello spazio di un processo, chiamata **stack**
- E' tuttavia possibile anche attaccare buffer presenti nella regione conosciuta col nome di **heap**

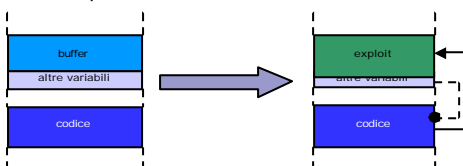
Heap-based overflow

- Inizialmente l'heap appare come segue



Heap-based overflow

- Una volta sovrascritto il buffer con il codice dell'exploit occorre dirottare il flusso di esecuzione all'interno del buffer modificato.
 - **IDEA:** sovrascrivere un puntatore al codice in modo da farlo puntare all'interno del buffer



Heap-based overflow

- Mostriamo un esempio di codice affetto da tale vulnerabilità

```
void g()
{
}

void main (int argc, char** argv)
{
    static struct {
        char buffer[10];
        void (*f)();
    } s;
    s.f = g;
    argv++;
    strcpy (&s.buffer, *argv);
    printf ("%08x %08xn", &s.f, &s.buffer);
    s.f();
}
```

Heap-based overflow

- Dando un argomento abbastanza lungo è possibile sovrascrivere il puntatore a funzione f e dirottare il flusso del programma

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
 - Buffer overflow classico
 - Frame pointer overflow
 - Heap-based overflow
 - Integer overflow
- ASN.1 vs BER
- Exploit per MS04-007

Integer overflow

- Si basano sulle caratteristiche dell'aritmetica dei calcolatori e sono le più difficili da sfruttare
- Prenderemo in esempio il linguaggio C in cui ciascun tipo di intero esistente è rappresentato con una variabile con un certo numero di bit
 - Esempio (su piattaforma Win32)
 - `int` è un intero a 32 bit
 - `short` è un intero a 16 bit

Integer overflow

- La seguente funzione è soggetta ad una vulnerabilità di integer overflow

```
int catvars(char* buf1, char* buf2,
            int len1, int len2)
{
    char buff[256];

    if ((len1 + len2) > 256)
        return -1;
    memcpy (buff, buf1, len1);
    memcpy (buff + len1, buf2, len2);
    ...
    return 0;
}
```

Integer overflow

- Il controllo effettuato nel test dell'if non tiene conto di un possibile *integer overflow*
 - il risultato dell'operazione $len1 + len2$ (che dovrebbe essere un intero, in quanto somma di due variabili intere), potrebbe essere un numero troppo grande (a causa dei valori addendi) per essere rappresentato con una variabile intera
 - ciò potrebbe portare ad una cattiva rappresentazione di tale valore somma che potrebbe superare il test, quando in realtà i buffer passati in argomento sono troppo grandi per essere memorizzati nel buffer `buff`
- In conclusione, alla funzione `memcpy` viene consentita la sovrascrittura dello stack, ricadendo così nel più classico *buffer overflow*

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- Exploit per MS04-007

ASN.1 vs BER

- Molti lavoratori nel campo della tecnologia dell'informazione sono spesso confusi circa cosa siano ASN.1 e BER, le differenze tra i due, o anche perché la distinzione sia importante
- L'**ASN.1** è una notazione astratta per la sintassi del trasferimento dati, su una rete, tramite messaggi, ecc...
 - indipendenza dalle piattaforme hardware
 - indipendenza dai linguaggi di programmazione
 - codifica dati selezionabile da chi invia e da chi riceve
 - indipendenza dei dati trasportati dalla loro implementazione
 - serializzabilità di dati complessi e multidimensionali

ASN.1 vs BER

- Oltre alla sintassi astratta, viene fornita anche una sintassi per codificare/decodificare e trasferire dati
- Per mappare la sintassi astratta in quella di trasferimento e viceversa, si usano le **BER** (Basic Encoding Rules)

ASN.1 vs BER

- Poiché ASN.1 è linguaggio degli standard, è comune trovare raccomandazioni di standard scritte in ASN.1
- La seguente è una lista di standard comunemente usati scritti in ASN.1:
 - X.400 (Electronic Messaging)
 - X.500 (Directory Services)
 - Request for Comments (RFC)
 - RFC 2251
 - Lightweight Directory Access Protocol (v3)
 - RFC 2252
 - Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions
 - RFC 2253
 - Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names
 - RFC 2254
 - The String Representation of LDAP Search Filters
 - RFC 2255
 - The LDAP URL Format
 - RFC 2256
 - A Summary of the X.500 User Schema for use with LDAPv3

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
 - ASN.1
 - BER
- Exploit per MS04-007

ASN.1

- ASN.1
 - standardizzata per la prima volta nel 1984 dalla CCITT (oggi ITU-T) sotto il nome di "Raccomandazione X.409"
 - in seguito, la ISO definisce due documenti separati
 - sintassi astratta (ASN.1)
 - regole di codifica (BER)
 - nel 1989, la CCITT pubblica le raccomandazioni X.208 e X.209
 - versione successiva ASN.1:1994 divisa in 4 parti
 - ISO 8824-1 | ITU-T X.680
 - Specification of basic notation
 - ISO 8824-2 | ITU-T X.681
 - Information object specification
 - ISO 8824-3 | ITU-T X.682
 - Constraint specification
 - ISO 8824-4 | ITU-T X.683
 - Parameterization of ASN.1 specifications

ASN.1

- Abstract Syntax Notation One, o **ASN.1**, è un linguaggio per definire standard senza far riferimento all'implementazione
- Per esempio, ASN.1 definisce
 - cosa è un "tipo"
 - cosa è un "modulo"
 - cosa è un INTERO
 - cosa è un BOOLEANO
 - ...

ASN.1

- In ASN.1 i tipi primitivi sono:
 - **BOOLEAN** TRUE, FALSE.
 - **INTEGER** Tutti i possibili numeri interi, senza delimitazione di rango.
 - **BIT STRING** Sequenze di bit di lunghezza non divisibile per otto.
 - **OCTET STRING** Sequenze di bytes.
 - **ENUMERATED**
 - **OBJECT IDENTIFIER** Identifica univocamente un nodo nell'albero di registrazione ISO, tramite il suo cammino dalla radice al nodo stesso.
 - **OBJECT DESCRIPTOR** Stringa di caratteri per l'identificazione di un nodo nell'albero di registrazione ISO, non è necessariamente unico.
 - **ANY** Equivale a VOID in C.
 - **EXTERNAL** Tipo di dati non descrivibili dai tipi di ASN-1.
 - **NULL** E' una macro che indica, in un tipo di dato assemblato, l'assenza di valore.

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- **ASN.1 vs BER**
 - ASN.1
 - BER
- Exploit per MS04-007

BER

- Insieme di regole per codificare dati ASN.1 in un flusso di ottetti che possono essere trasmessi su un link di comunicazione
 - definito nelle raccomandazioni ITU-T X.209 e X.690
- Altri metodi per codificare dati ASN.1 sono:
 - Distinguished Encodig Rules (**DER**)
 - simile a BER, usato spesso per trasferire certificati
 - Canonical Encoding Rules (**CER**)
 - Packing Encoding Rules (**PER**)
 - permette una forte compressione dei dati

BER

- BER definisce:
 - Metodi per codificare valori ASN.1.
 - Regole per decidere quando usare un dato metodo.
 - Il formato di specifici ottetti nei dati.
- Con BER i tipi vengono codificati nel formato **TLV (tag / lunghezza / valore)**
 - Un po' inefficiente perché in alcuni casi vengono inserite informazioni ridondanti (e.g. un 'Integer 32' è ovviamente lungo 32 bit, quindi il campo **lunghezza** è ridondante).

ASN.1 vs BER

- Fondamentalmente la differenza tra ASN.1 e BER può essere chiarita con la seguente similitudine

ASN.1 è come un linguaggio di programmazione,
mentre BER è come un compilatore per quel linguaggio

ASN.1 vs BER

- Per chiarire meglio quanto detto finora, consideriamo il seguente esempio
 - quanto segue è la descrizione informale di un record "PersonnelRecord"

```
Name:          John P Smith
Date of Birth: 17 July 1959
...
(altri dati)
```

ASN.1 vs BER

- La descrizione ASN.1 del suddetto record (quello standard) potrebbe essere:

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
  Name,
  title [0]          VisibleString,
  dateOfBirth [1]   Date,
  (altri tipi definiti)
}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
  givenName      VisibleString,
  initial        VisibleString,
  familyName     VisibleString
}
```

ASN.1 vs BER

- In seguito, l'applicazione mappa i dati del personale nella struttura PersonnelRecord (formato dei dati ASN.1) e poi applica le Basic Encoding Rules (BER) ai dati ASN.1.

ASN.1 vs BER

- Questo è come dovrebbe apparire (con l'eccezione che i nomi dovrebbero essere convertiti in ASCII):

Personnel Record	Length	Contents																																										
60	8185																																											
		<table border="1"><thead><tr><th>Name</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td></td><td>61</td><td></td></tr><tr><td></td><td></td><td><table border="1"><thead><tr><th>VisibleString</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>1A</td><td>04</td><td>"John"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>01</td><td>"P"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>05</td><td>"Smith"</td></tr></tbody></table></td></tr><tr><td>DateofBirth</td><td></td><td></td></tr><tr><td>A0</td><td>0A</td><td></td></tr><tr><td></td><td></td><td><table border="1"><thead><tr><th>Date</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>43</td><td>08</td><td>"19590717"</td></tr></tbody></table></td></tr></tbody></table>	Name	Length	Contents		61				<table border="1"><thead><tr><th>VisibleString</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>1A</td><td>04</td><td>"John"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>01</td><td>"P"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>05</td><td>"Smith"</td></tr></tbody></table>	VisibleString	Length	Contents	1A	04	"John"	VisibleString	Length	Contents	1A	01	"P"	VisibleString	Length	Contents	1A	05	"Smith"	DateofBirth			A0	0A				<table border="1"><thead><tr><th>Date</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>43</td><td>08</td><td>"19590717"</td></tr></tbody></table>	Date	Length	Contents	43	08	"19590717"
Name	Length	Contents																																										
	61																																											
		<table border="1"><thead><tr><th>VisibleString</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>1A</td><td>04</td><td>"John"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>01</td><td>"P"</td></tr><tr><td>VisibleString</td><td>Length</td><td>Contents</td></tr><tr><td>1A</td><td>05</td><td>"Smith"</td></tr></tbody></table>	VisibleString	Length	Contents	1A	04	"John"	VisibleString	Length	Contents	1A	01	"P"	VisibleString	Length	Contents	1A	05	"Smith"																								
VisibleString	Length	Contents																																										
1A	04	"John"																																										
VisibleString	Length	Contents																																										
1A	01	"P"																																										
VisibleString	Length	Contents																																										
1A	05	"Smith"																																										
DateofBirth																																												
A0	0A																																											
		<table border="1"><thead><tr><th>Date</th><th>Length</th><th>Contents</th></tr></thead><tbody><tr><td>43</td><td>08</td><td>"19590717"</td></tr></tbody></table>	Date	Length	Contents	43	08	"19590717"																																				
Date	Length	Contents																																										
43	08	"19590717"																																										

ASN.1 vs BER

- Alla fine ciò che viene realmente trasmesso è la sequenza

```
60 81 85 61 10 1A 04 ...
... 0A 43 08 19 59 07 17
```

Overview

- Introduzione
- Descrizioni tecniche
- Buffer overflow ed exploit
- Vulnerabilità di buffer overflow ed attacchi
- ASN.1 vs BER
- [Exploit per MS04-007](#)

Un exploit per MS04-007

- Il 14 febbraio 2004 è stato reso disponibile un exploit DOS che sfrutta la falla MS04-007
 - [MS04-007-dos.c](#)
 - creato dal 23enne Cristophe Devine
 - causa l'interruzione del funzionamento del sistema operativo (Denial Of Service, ovvero DOS)
 - ma non permette l'esecuzione di programmi nocivi

