HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

NASL - Nessus Attack Scripting Language

CCERT-PUBDOC-2005-03-114

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.

Rezultat toga rada ovaj je dokument koji je nastao suradnjom CARNet CERT-a i LS&S-a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, **www.cert.hr** - nacionalno središte za sigurnost računalnih mreža i sustava.

LS&S, **www.lss.hr -** laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

Provjera ranjivosti (engl. *vulnerability scanning*) postupak je kojim se sigurnosni stručnjaci služe kako bi pravovremeno uočili, a nakon toga i uklonili sigurnosne propuste u informacijskim sustavima. Metode koje se pri tome koriste najčešće se baziraju na onima koje neovlašteni korisnici inače koriste u sklopu provođenja malicioznih aktivnosti. Kako se radi o jednostavnom, a prilično učinkovitom postupku, riječ je o jednoj od najčešće primjenjivanih tehnika podizanja razine sigurnosti računalnih sustava.

U posljednje vrijeme tržište sigurnosnih alata preplavljeno je velikim brojem programskih alata za provjeru ranjivosti na lokalnim (engl. *local*) i udaljenim (engl. *remote*) računalima. Između njih svakako se ističe Nessus, koji danas s pravom nosi titulu najpopularnijeg alata za provjeru sigurnosti. Razlog Nessusove popularnosti je prije svega besplatna distribucija i korištenje alata (iako je od 1. siječnja 2005. godine uveden novi način licenciranja po kojem se određeni moduli naplaćuju), uz mogućnosti i performanse koje često nadmašuju velik dio komercijalnih i iznimno skupih alata. Nadalje, velik broj korisnika, koji aktivno sudjeluju na razvoju Nessus projekta, osigurava brzo dodavanje novih funkcionalnosti te redovito objavljivanje modula za ispitivanje novootkrivenih sigurnosnih propusta.

U dokumentu su ukratko opisane osnovne značajke i način rada Nessus programskog paketa, a najveći dio posvećen je NASL programskom jeziku, alatu koji korisnicima omogućuje jednostavnu i brzu izradu modula za provjeru ranjivosti korištenjem Nesssus programskog paketa. Opisana je osnovna namjena, sintaksa te neke od važnijih funkcija NASL jezika, a dane su i neke općenite smjernice za izradu NASL skripti.

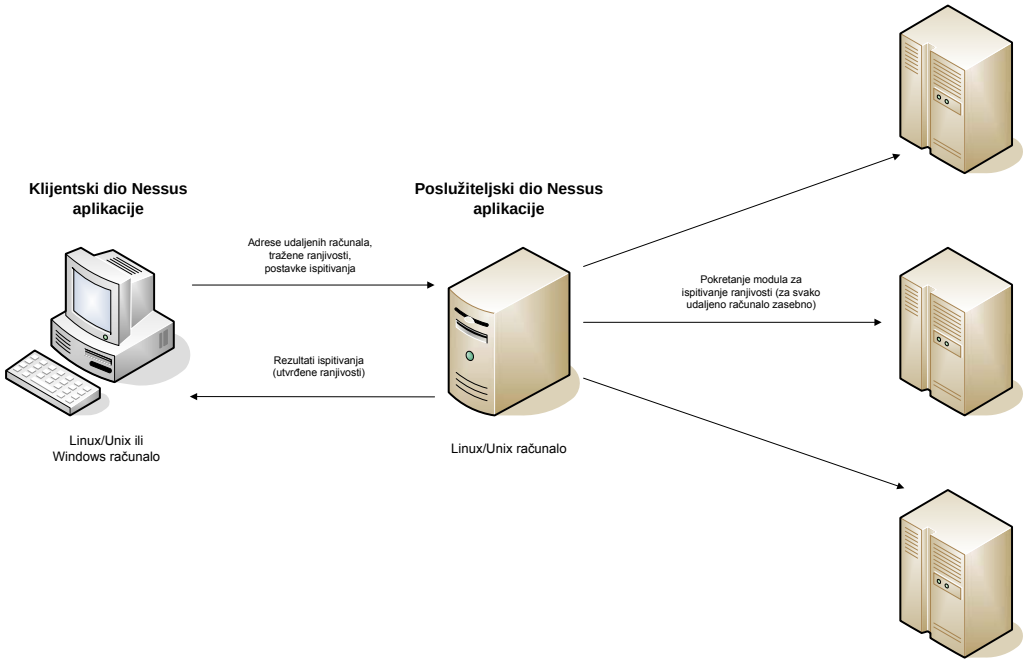
2. Nessus programski paket

Nessus programski alat za provjeru ranjivosti računalnih sustava bazira se na klijent-poslužitelj modelu. Za razliku od klijentskog dijela koji je dostupan za Windows i Linux operacijske sustave, poslužiteljski dio pokreće se isključivo na računalu s Linux/Unix operacijskim sustavom, što se može navesti kao jedan od rijetkih nedostataka ovog iznimno kvalitetnog programskog paketa.

Također, kako je riječ o programu koji nema prevelike zahtjeve za resursima, računalo na kojem se pokreće ne mora biti suviše jakih performansi (Pentium III sa 512MB radne memorije zadovoljiti će potrebe većine korisnika). Dodamo li tome činjenicu da je Linux operacijski sustav besplatan, očito je da postavljanje Nessus poslužitelja predstavlja isplativo ulaganje, pogotovo ukoliko se u obzir uzmu sve prednosti koje program nudi.

Klijentski dio aplikacije predstavlja korisničko sučelje za upravljanje Nessus poslužiteljem i kako je već ranije spomenuto dostupan je i za Linux i za Windows operacijske sustave. Kod Linux operacijskih sustava klijentski dio aplikacije moguće je koristiti ili putem grafičkog sučelja ili putem naredbenog retka, ovisno o navikama i potrebama korisnika.

Poslužiteljski dio Nessus programa pokreće se kao pozadinski proces (engl. *daemon*) koji na TCP/1241 mrežnom portu osluškuje zahtjeve klijenata. Nakon uspostave komunikacije i autentikacije korisnika, putem klijentskog dijela aplikacije potrebno je definirati parametre postupka provjere ranjivosti (IP adrese računala koja je potrebno testirati, ranjivosti koje se ispituju, te niz drugih postavki kao je što su, primjerice, maksimalno vrijeme testiranja, broj ponovnih pokušaja u slučaju neuspjeha i sl.). Nakon definiranih parametara provjere i pokretanja postupka, poslužiteljski dio odgovoran je za provođenje svih odabranih testova te slanje rezultata klijentskom dijelu aplikacije. Shematski prikaz opisanog postupka prikazan je na sljedećoj slici (Slika 1).



Slika 1. Shematski prikaz postupka provjere ranjivosti pomoću Nessus alata

Jedna od najvećih prednosti Nessus programa je njegova modularnost, koja korisnicima omogućuje da sami odabiru koje će pojedinačne testove Nessus provoditi, odnosno koje će module za ispitivanje ranjivosti pokretati. Također, korisnici mogu sami razvijati i dodavati vlastite module što im omogućuje jednostavno kreiranje specifičnih testova koji inicijalno nisu dostupni unutar Nessus programa.

Na primjer, ukoliko sigurnosna politika tvrtke zahtjeva da se umjesto nesigurnog Telnet servisa na svim računalima koristi SSH servis, mrežni administrator može jednostavno izraditi vlastiti modul koji će u vrlo kratkom vremenu provjeriti sva računala na sustavu i prijaviti sva odstupanja od definirane sigurnosne politike. Na sličan način moguće je detektirati i neovlašteno postavljene *peer-to-peer* programe na osobnim računalima korisnika, a moguće su i mnogo složenije primjene

Module za provjeru ranjivosti teoretski bi bilo moguće pisati u mnogim skriptnim jezicima kao što su npr. Perl, Pyton i sl., no takvi moduli imali bi određene nedostatke koji bi mogli utjecati na fleksibilnost programa. Prvo, kod većine spomenutih jezika potrebno je instalirati dodatnu podršku ukoliko se žele koristiti neke dodatne funkcionalnosti. U Perl-u, na primjer, nije moguće kreirati i slati IP pakete ukoliko nije instaliran `Net::RawIP` modul, a to je samo jedan od primjera. Također, prilikom prijenosa skripti s jednog operacijskog sustava na drugi, može doći i do problema s kompatibilnošću što svakako nije zanemariv problem. Nadalje, mnogi skriptni jezici imaju prilično velike zahtjeve nad resursima računala što bi moglo utjecati na performanse programa prilikom provjere većeg broj računala. Konačno, korištenje tuđih skripti može biti problematično i sa stanovišta sigurnsti ukoliko korisnik sam nije dobar poznavatelj jezika u kojem je skripta napisana.

Kako bi se riješili opisani problemi, za pisanje Nessus modula osmišljen je poseban skriptni jezik pod nazivom NASL (engl. *Nessus Attack Scripting Language*). NASL je skriptni jezik razvijen isključivo za potrebe programiranja modula za provjeru ranjivosti korištenjem Nessus programa. Prilikom osmišljavanja NASL jezika osnovna ideja bila je razviti jednostavan i lako razumljiv skriptni jezik, koji će biti maksimalno prilagođen izradi modula za provjeru ranjivosti i koji će korisnicima omogućiti jednostavnu izradu i razmjenu modula, ne vodeći pritom brigu o operacijskim sustavima, sigurnosnim problemima i sl.

Neke od prednosti NASL jezika u odnosu na ostale skripte jezike navedene su u nastavku:

- instalacija NASL-a je jednostavna, a za pisanje modula nije potrebno instalirati dodatne pakete;
- NASL jezik je u potpunosti prilagođen Nessus programskom paketu – izrada modula za Nessus je brzo i jednostavno;
- NASL ne postavlja velike zahtjeve za resursima;
- gramatika jezika vrlo je slična C programskom jeziku, tako da većini programera savladavanje jezika ne predstavlja poseban problem;
- moduli napisani u NASL jeziku su prenosivi i lako se prilagođavaju specifičnim potrebama korisnika;
- moduli napisani u NASL jeziku prihvatljivi su sa stanovišta sigurnosti – NASL garantira da tijekom izvođenja skripte neće doći do:
    - izvođenja naredbi na lokalnom računalu;
    - slanja paketa bilo kojem udaljenom računalu osim onoga koje je podvrgnuto testiranju.

Naravno, potrebno je istaknuti da je upravo zbog svoje jednostavnosti i orijentiranosti na rješavanje jednog tipa problema, NASL bitno ograničen u odnosu na skriptne jezike opće namjene. Najvažnija ograničenja NASL jezika su:

- NASL podržava samo jednostavne tipove podataka (tako, primjerice, ne podržava brojeve s pomičnom točkom ili strukture);
- programski jezik NASL prilično je ograničen u odnosu na ostale skriptne jezike (pri tome se prije svega misli na broj i mogućnosti ugrađenih funkcija);
- NASL je sporiji od mnogih skriptnih jezika, a raspolaganje memorijom (engl. *memory management*) nije optimalno riješeno;
- za sada ne postoji NASL alat za otkrivanje i uklanjanje grešaka (engl. *debugger)*, no NASL prevodilac (engl. *interpreter*) upozorava na pogreške prilikom izvođenja skripte.

Navedena ograničenja čine NASL nepogodnim za razvoj programa druge namjene, no prilikom pisanja modula za ispitivanje ranjivosti ne predstavljaju veći problem.

U nastavku dokumenta detaljnije je analiziran NASL programski jezik i postupak pisanja Nessus modula u tom jeziku.

3. Gramatika jezika NASL

Velik dio gramatike NASL jezika preuzet je od danas najraširenijeg programskog jezika – C-a. Ideja ovakvog pristupa bila je maksimalno skratiti vrijeme koje je korisniku potrebno za savladavanje gramatike jezika te izradu vlastitih modula za provjeru ranjivosti. No, s ciljem pojednostavljenja, iz NASL-a su izbačeni mnogi dijelovi C-a (primjerice strukture i objekti, pokazivači itd.) koji nisu nužni za realizaciju Nessus modula.

U nastavku je dan kratki pregled gramatike s posebnim osvrtom na one dijelove koji se bitnije razlikuju u odnosu na C programski jezik.

3.1. Naredbe i komentari

Kao i u C programskom jeziku, svaka naredba u NASL-u završava znakom `;`. Jednostavna naredba pridruživanja izgleda ovako:

```
a = 1;
```

Znak za početak komentara je `#`. Tekst koji se nađe između navedenog znaka i kraja retka interpreter će ignorirati prilikom prevođenja programa. Primjeri ispravnih komentara su:

```
# Ovo je komentar...
a = 1;            # ...bas kao i ovo
```

Važno je naglasiti da u NASL jeziku ne postoji mogućnost komentiranja više redaka odjednom.

3.2. Tipovi podataka

U NASL jeziku postoje sljedeća četiri tipa podatka:

- **integer** – cijeli broj;
- **string** – znakovni niz;
- **array** – polje;
- vrijednost `NULL`.

Za dohvat pojedinog elementa polja ili određenog znaka u znakovnom nizu koristi se operator `[]`. Kao i u C-u, prvi član polja odnosno prvi znak u nizu ima indeks **0**. Osim ovog uobičajenog načina indeksiranja, u NASL-u je članove polja moguće indeksirati i znakovnim nizovima.

`NULL` je vrijednost koja se pridjeljuje neinicijaliziranoj varijabli ukoliko se takva pokuša koristiti u programu. Također, ugrađene funkcije vraćaju `NULL` vrijednost u slučaju kritične pogreške.

U NASL-u ne postoji **boolean** tip podataka. Operatori usporedbe vraćaju cjelobrojne vrijednosti – vrijednost **0** ukoliko uvjet nije zadovoljen, te vrijednost **1** ukoliko je uvjet zadovoljen.

Operandi logičkih operacija mogu biti svi tipovi podataka. Pri tome, `NULL` vrijednost (neinicijalizirana vrijednost) predstavlja neistinu. Cijeli brojevi predstavljaju neistinu ukoliko su jednaki **0**, istinu inače. Znakovni nizovi su neistiniti ukoliko su prazni, istiniti inače. Polja su uvijek istinita, čak i ukoliko ne sadrže niti jedan član.

3.2.1. Konstante

Brojčane konstante mogu se pisati u decimalnom, heksadecimalnom, oktanom i binarnom obliku, pri čemu je način pisanja preuzet iz C-a. Broj **1234**, primjerice, može se zadati na sljedeće načine:

```
a = 1234               # decimalni zapis
a = 0x4D2              # heksadecimalni zapis
a = 02322              # oktalni zapis
a = 0b010011010010     # binarni zapis
```

Znakovni nizovi (engl. *string*) zadaju se na dva načina – kao "čisti" (engl. *pure*) i "nečisti" (engl. *impure*). "Čisti" znakovni nizovi omeđuju se jednostrukim navodnicima (znak `'`), dok se "nečisti" omeđuju dvostrukim navodnicima (znak `"`).

Razlika između ova dva načina zadavanja jest u značenju znaka `\`. U "čistim" znakovnim nizovima ovaj znak predstavlja prekidni znak (engl. *escape character*) koji se upotrebljava za kodiranje neispisivih znakova (engl. *nonpritable characters*) i navodnika. U "nečistim" znakovnim nizovima znak `\` se ne tumači kao posebni znak. U tablici 1.1. dan je pregled kodova preuzetih iz C-a i njihovog značenja.

Kod	Značenje
\0	*Nul character*
\n	Novi redak (*new line*)
\t	Vodoravni odmak (*horizontal tabulation*)
\v	Okomiti odmak (*vertical tabulation*)
\r	*Line feed*
\f	*Form feed*
****	Obrnuta kosa crta (*backslash*)
\'	Jednostruki navodnik (*single quote*)
\"	Dvostruki navodnik (*double quote*)
\x42	Znak čiji je ASCII kod $66=42_{16}$ – slovo **b**

Tablica 1: Posebni znakovi

Za unos posebnih znakova u znakovni niz može se koristiti funkcija `string()` o kojoj će više riječi biti kasnije, u poglavlju o funkcijama za rukovanje znakovnim nizovima. Primjeri ispravno napisanih znakovnih nizova:

```
a = "Glavni grad Hrvatske je\nZagreb";
# a poprima vrijednost: "Glavni grad Hrvatske je\nZagreb"

b = 'Glavni grad Hrvatske je\nZagreb'
# b poprima vrijednost: "Glavni grad Hrvatske je Zagreb"

c = string("Glavni grad Hrvatske je\nZagreb");
# c poprima vrijednost: "Glavni grad Hrvatske je Zagreb"
```

U NASL jeziku postoji veći broj ugrađenih (predefiniranih) konstanti. Primjerice, radi intuitivnijeg pisanja logičkih izraza definirane su konstante `FALSE` (ima vrijednost **0**) i `TRUE` (ima vrijednost **1**). Cjelovit popis istih moguće je naći u dokumentaciji skriptnog jezika NASL (Poglavlje 8).

3.2.2. Varijable

Varijable u NASL-u nije nužno deklarirati prije upotrebe, osim u slučaju deklaracije lokalnih varijabli u korisnički definiranim funkcijama, kada se želi izbjeći promjena istoimenih globalnih varijabli. NASL prevodilac vodi brigu o tipovima varijabli, pa će tako u slučaju pogrešnog povezivanja prijaviti pogrešku.

Primjer deklaracije varijable:

```
local_var var;
global_var var;
```

Programer ne mora voditi niti brigu oko zauzimanja memorije, jer se ona zauzima i oslobađa dinamički, o čemu brigu također vodi prevodilac.

3.3. Operatori

Kao što je već napomenuto, operator pridruživanja je znak `=`. Kao i u C-u naredbe pridruživanja moguće je pisati skraćeno. Na primjer, naredba kojom se varijabla povećava za **1** može se skraćeno pisati:

```
a += 1;                    # isto kao i: a = a + 1;
```

Aritmetički operatori zbrajanja, oduzimanja, množenja, dijeljenja i ostatka cjelobrojnog dijeljenja – redom `+`, `-`, `*`, `/`, `%` – preuzeti su iz C-a. No, u NASL-u operatori `+` i `-` mogu se koristiti i za

manipulaciju znakovnim nizovima. Znak + u tom slučaju predstavlja spajanje dva niza, dok se znak – koristi za uklanjanje podniza iz znakovnog niza. Sljedeći primjer ilustrira rad tih dvaju operatora:

```
a = "grad ";
b = "Zagreb je najve•i grad u Hrvatskoj";

c = a + b;        # c poprima vrijednost: "grad Zagreb
                  # je najve•i grad u Hrvatskoj";
d = b - a;        # d poprima vrijednost: "Zagreb je
                  # najve•i u Hrvatskoj";
```

Iz C-a je preuzeta i većina logičkih operacija na razini bita. Operatori ~, |, & i ^ vrše redom logičke operacije negacije, disjunkcije, konjunkcije na razini bita:

```
a = 0b0011;
b = 0b0101;

c = ~a;           # c poprima vrijednost 0b1100
d = a | b;        # d poprima vrijednost 0b0111
e = a & b;        # e poprima vrijednost 0b0001
f = a ^ b;        # f poprima vrijednost 0b0110
```

U NASL-u postoje i operatori za posmak bitova, vrlo slični onima u C-u. Operatori za posmak su: << (logički posmak ulijevo), >> (aritmetički posmak udesno) i >>> (logički posmak udesno). Kao primjer upotrebe ovih operatora može poslužiti dijeljenje varijable s brojem **4**, koje je moguće izvesti pomoću aritmetičkog posmaka udesno:

```
b = a<<2;                # b poprima vrijednost a/4
```

Za gradnju uvjeta koriste se operatori uspoređivanja – = = (jednako), != (različito), < (manje od), > (veće od), <= (manje od ili jednako) i >= (veće od ili jednako) – te logički operatori – ! (unarni operator negacije), || (binarni operator disjunkcije) i && (binarni operator konjunkcije). Uspoređivati se mogu cijeli brojevi i znakovni nizovi. Prilikom uspoređivanja znakovnih nizova pojedini znakovi uspoređuju se prema pripadajućim ASCII kodovima.

Osim svih dosada navedenih, u NASL-u postoje i neki specifični operatori. Prvi je par *boolean* operatora >< i >!< koji služe za pronalaženje podniza u znakovnom nizu. Izraz će poprimiti vrijednost `true` ako je znakovni niz naveden **ispred** operatora >< sadržan u znakovnom nizu navedenom **iza** istog operatora. Rad ovog operatora moguće je ilustrirati na znakovnim nizovima iz prethodnog primjera:

```
a >< b
b >< a
a >< d
```

Prvi od izraza navedenih u gornjem primjeru evaluirati će se kao istinit (`true`), a druga dva kao neistiniti (`false`). Operator >!< vraća negaciju rezultata koji vraća operator ><.

Nadalje, postoji operatori =~ (*regex match*) i !~ (*regex don't match*). Operator =~ vraća isti rezultat kao i funkcija `ereg()`, o kojoj će biti riječi kasnije, dok operator !~ vraća suprotan rezultat.

Drugi operator specifičan za NASL jest operator x. Ovaj operator služi za ponavljanje poziva funkcije određeni broj puta. Tako će, primjerice, naredba

```
send_packet(p) x 10;
```

pozvati funckciju `send_packet` deset puta. Isto bi se moglo učiniti i korištenjem programskih petlji (opisanih u sljedećem poglavlju), no korištenjem operatora x moguće je značajno poboljšati brzinu rada skripte.

3.4. Naredbe za kontrolu toka

Za kontrolu toka programa u NASL jeziku koristi se jednostavna naredba grananja `if` te petlje `while`, `repeat`, `for` i `foreach`.

Opći oblik `if` naredbe je:

```
if ( <uvjet> )
    <blok>
[ else
    <blok> ]
```

pri čemu `<blok>` čini jedna naredba (iza koje, naravno, slijedi znak `;`) ili niz naredbi omeđenih zagradama:

```
{
    # niz naredbi
}
```

Ukoliko je uvjet zadan u zagradi na početku naredba ispunjen, izvršava se prvi blok naredbi, a ukoliko nije, izvršava se drugi blok naredbi. `else` dio naredbe nije obavezan, te ukoliko isti ne postoji u slučaju da uvjet nije zadovoljen neće biti izvršena niti jedna naredba.

Opći oblik `while` petlje je:

```
while ( <uvjet> )
    <blok>
```

Prilikom svakog ulaska u petlju, provjerava se je li uvjet ponavljanja ispunjen. Ukoliko je, izvršavaju se naredbe iz petlje te se tijek programa usmjerava na provjeru uvjeta.

Opći oblik `repeat` petlje je:

```
repeat
    <blok>
until ( <uvjet> );
```

Za razliku od prethodne petlje, blok naredbi unutar `repeat` petlje ponavlja se ukoliko uvjet **nije** zadovoljen. Druga bitna razlika između ove i `while` petlje jest to što se uvjet provjerava nakon prolaska kroz petlju, što znači da će blok naredbi unutar `repat` petlje sigurno biti izvršen bar jednom (što ne mora biti slučaj kod `while` petlje).

Opći oblik `for` petlje je:

```
for ( <izraz1>; <uvjet>; <izraz2> )
    <blok>
```

Prvi izraz naveden u zagradi (inicijalizacija) izvršava se jednom, i to prije prvog ulaska u petlju. Niz naredaba u petlji izvršava se ukoliko je uvjet ponavljanja ispunjen. Posljednji izraz u zagradi izvršava se nakon svakog prolaska kroz petlju. Nakon izvršavanja naredbi unutar petlje tijek programa se usmjerava na provjeru uvjeta.

Petlja `foreach` koristi se za iteriranje kroz članove polja. Opći oblik ove petlje je:

```
foreach var (<polje>)
    <blok>
```

Blok naredbi ponavlja se po jedan put za svaki član polja. Vrijednost `var` poprima u svakom prolazu kroz petlju vrijednost odgovarajućeg člana polja.

Za prijevremeni izlaz iz petlje koristiti se naredba `break;`, dok za preskakanje ostatka naredbi iz bloka i skok na provjeru uvjeta služi naredba `continue;`.

3.5. Korisnički definirane funkcije

Osim korištenja ugrađenih funkcija, korisnik može definirati i vlastite funkcije koje će koristiti u programu. Deklaracija korisnički definirane funkcije u NASL-u ima sljedeći oblik:

```
function <ime_funkcije>( [argument1 [, argument2 [, ...]]] )
```

```
{
  #
  # Naredbe koje •ine tijelo funkcije
  #
  [ return <povratna_vrijednost>; ]
}
```

Funkcija ne mora vraćati vrijednost, no ukoliko je to slučaj koristi se ključna riječ `return`. Bitna razlika u odnosu na C programski jezik je to što `return` nije prava ključna riječ, već zamjenjuje poziv funkcije, tako da se povratna vrijednost može pisati i u zagradama.

Rekurzivni pozivi funkcija su dopušteni, no korisnički definirane funkcije ne smiju sadržavati druge korisničke funkcije (ugnježđivanje funkcija nije dozvoljeno). Korisničke funkcije smiju pozivati druge korisničke funkcije, čak i ukoliko je deklaracija pozvane funkcije niže u kodu.

3.5.1. Argumenti funkcija

Funkcije bez argumenata u NASL-u se pozivaju kao i C-u, no pozivanje funkcija s argumentima riješeno je na nešto drugačiji način. Argumenti funkcija u NASL-u mogu biti:

- **imenovani** – argumenti koji su deklarirani prilikom definiranja funkcije;
- **neimenovani (anonimni)** – argumenti koji nisu deklarirani prilikom definiranja funkcije.

Prilikom poziva funkcije, uz vrijednost imenovanog argumenta potrebno je istaknuti i njegovo ime. Redoslijed kojim se zadaju vrijednosti imenovanih argumenata nije bitan, a nije potrebno zadati ni vrijednosti svih imenovanih argumenata.

Na primjer, sljedeća dva poziva funkcije sasvim su identična i ispravna, osim što u drugom pozivu nije zadana vrijednost imenovanog argumenta `c` (unutar funkcije ona će biti `NULL`):

```
f(a: 1, b: "ABC", c: 2, "Bjelovar", 1234);
f("Bjelovar", b: "ABC", 1234, a: 1);
```

Vrijednost imenovanog argumenta iz funkcije se dohvaća preko imena. Vrijednosti neimenovanih argumenata dohvaćaju se preko polja `_FCT_ANON_ARGS`. Potrebno je naglasiti da u starijim inačicama NASL interpretera ova tehnika nije podržana te poziv ove varijable vraća vrijednost `NULL`. Nadalje, važno je istaknuti da preko polja `_FCT_ANON_ARGS` nije moguć dohvat vrijednosti imenovanih argumenata, niti je u polje moguće pisati.

Sljedeći primjer opisuje definiciju jednostavne funkcije za zbrajanje. Brojevi koje se želi zbrojiti funkciji se prosljeđuju preko neimenovanih argumenata. Uz to, funkcija ima i imenovani argument `verbose`. Vrijednost ovog argumenta nije nužno zadati prilikom poziva funkcije, no ukoliko je ona zadana i različita od **0**, funkcija `zbroji` će ispisati izračunato rješenje na zaslon.

```
function zbroji(verbose)
{
  local_var i, zbroj;
  zbroj = 0;
  for (i = 0; _FCT_ANON_ARGS[i]; i++)
    zbroj += _FCT_ANON_ARGS[i];
  if (verbose)
    display("zbroj = ", zbroj, "\n");
  return zbroj;
}
```

Slijede primjeri ispravnih poziva funkcije `zbroji`:

```
a = zbroji(3, 5, 6);
    # varijabli a se pridružuje vrijednost 14, nema ispisa
zbroji(2, 4, verbose: 1);
    # ispisuje se: zbroj = 6
```

4. Ugrađene funkcije

Biblioteka ugrađenih funkcija NASL programskog paketa sadržava velik broj funkcija. U nastavku je dan kratki pregled važnijih funkcija dok je cjelovit popis funkcija i pripadajućih parametara moguće naći u dokumentaciji navedenoj na kraju dokumenta (Poglavlje 8).

4.1. Funkcije opće namjene

Funkcije opće namjene podrazumijevaju funkcije za kontrolu toka programa, funkcije za provjeru tipa podataka i baratanje složenim tipovima podataka, vremenske funkcije i sl. Najčešće korištene funkcije ukratko su opisane u nastavku.

4.1.1. Funkcija `exit()`

Ova funkcija služi za prekid izvođenja programa. Kao jedini argument uzima kôd koji označava uzrok prekida (engl. *error code*). U slučaju normalnog prekida programa funkciji se predaje vrijednost **0**:

```
exit(0);
```

4.1.2. Funkcija `display()`

Funkcija `display()` uzima neodređeni broj neimenovanih argumenata te ih ispisuje na zaslon. Prije ispisa na zaslon, svaki se argument pojedinačno pozivom funkcije `string()` (vidi sljedeće poglavlje) pretvara u znakovni niz.

4.1.3. Funkcija `isnull()`

Provjerava da li je neimenovani argument inicijaliziran ili ne. Ukoliko je argument inicijaliziran funkcija vraća vrijednost `FALSE`, inače `TRUE`.

4.1.4. Funkcija `make_array()`

Funkcija uzima paran broj neimenovanih argumenata te od njih stvara polje. Pri tome svaki par argumenata predstavlja par *indeks-vrijednost* odgovarajućeg člana polja.

Na primjer, sljedeći poziv funkcije:

```
polje = make_array(1, 'Jedan', 'Dva', 2);
```

čini isto što i par naredbi:

```
polje[1] = 'Jedan';
polje['Dva'] = 2;
```

4.1.5. Funkcije `sleep()` i `usleep()`

Funkcije kao argument prihvaćaju jednu cjelobrojnu vrijednost. Funkcija `sleep()` čeka zadani broj sekundi, a funkcija `usleep()` zadani broj milisekundi.

4.1.6. Funkcija `rand()`

Funkcija `rand ()` koristi se za generiranje slučajnih brojeva. Primjer korištenja dan je u nastavku:

```
id = rand() % 65535;    # id poprima slu•ajnu vrijednost u
                        # rasponu 0 do 65534
```

4.2. Funkcije za rad sa znakovnim nizovima

Tijekom ispitivanja sigurnosnih propusta često se javlja potreba za pretraživanjem i uspoređivanjem znakovnih nizova (primjerice, utvrđivanje inačice servisa iz tzv. *banner*-a i sl.). Upravo je zato u NASL ugrađen određen broj prilično snažnih funkcija za baratanje znakovnim nizovima. U nastavku je dan pregled važnijih.

4.2.1. Funkcija `string()`

Funkcija `string()` uzima neodređeni broj neimenovanih argumenata, svaki zasebno pretvara u čisti znakovni niz te dobivene nizove spaja u konačni niz koji se vraća kako rezultat. Prilikom pretvaranja argumenata u znakovne nizove vrijede pravila:

- "čisti" znakovni nizovi se ne mijenjaju;
- "nečisti" znakovni nizovi pretvaraju se u "čiste";
- cijeli brojevi pretvaraju se u znakovni zapis u dekadskoj bazi;
- polja se pretvaraju u znakovne nizove član po član;
- nedefinirane (`NULL`) vrijednosti se ignoriraju.

Primjer korištenja `string()` funkcije:

```
broj = 900000;
grad = "Zagreb";

a = string('Grad ', grad, "ima:\n", broj+1, 'stanovnika');
      # a poprima vrijednost:
      # "Grad Zagreb ima" (prelazak u novi red)
      # "900001 stanovnika"
```

4.2.2. Funkcija `egrep()`

Funkcija `egrep()` pretražuje zadani znakovni niz liniju po liniju, tražeći linije koje odgovaraju danom uzorku.

Parametri funkcije `egrep()` su:

- `pattern` – traženi uzorak, zadan u *standard extended POSIX regex* formatu (vidi `man 1 egrep` za detalje);
- `string` – znakovni niz u kojemu se traži uzorak;
- `icase` – (inicijalno `FALSE`) ukoliko je vrijednost ovog argumenta `TRUE` ignorirati će se razlika malih i velikih slova.

Funkcija kako rezultat vraća znakovni niz dobiven povezivanjem svih linija koje odgovaraju uzorku.

4.2.3. Funkcija `ereg()`

Slično kao i prethodnom primjeru, funkcija `ereg ()` služi za pronalaženje određenog uzorka unutar zadanog znakovnog niza.

Parametri funkcije `ereg()` su:

- `pattern` – traženi uzorak;
- `string` – znakovni niz u kojemu se traži uzorak;
- `icase` – (inicijalno `FALSE`) ukoliko je vrijednost ovog argumenta `TRUE` ignorirati će se razlika malih i velikih slova;
- `multiline` – (inicijalno `FALSE`) ukoliko je vrijednost ovog argumenta `TRUE` pretraga će se nastaviti i nakon prvog znaka za prelaz u novi red.

Funkcija vraća prvu pojavu traženog uzorka u znakovnom nizu.

4.3. Funkcije za rukovanje mrežnim paketima

Ispitivanje sigurnosnih propusta na udaljenom računalu često zahtjeva kreiranje specifičnih mrežnih paketa (nerijetko čak i neispravnih). U NASL jezik ugrađene su stoga prilično moćne funkcije za kreiranje IP paketa, TCP i UDP paketa. Također, postoje i funkcije za kreiranje ICMP I IGMP paketa, no one imaju nešto slabije mogućnosti (npr. nije moguće zadati vrijednosti svih polja paketa).

4.3.1. Funkcije za odbacivanje paketa

Neželjene pakete moguće je odbaciti korištenjem funkcija `dump_ip_packet()`, `dump_tcp_packet()` i `dump_udp_packet()`. Funkcije preko argumenata primaju neograničen broj paketa odgovarajućeg tipa i ne vraćaju nikakvu vrijednost.

4.3.2. Rukovanje IP paketima

Za kreiranje IP paketa koristi se funkcija `forge_ip_packet()`. Funkcija prima sljedeće imenovane parametre:

- `data`: podaci koji se prenose paketom (engl. *Data*);
- `ip_v`: verzija IP protokola (engl. *Version*) – inicijalno **4**;
- `ip_hl`: duljina zaglavlja IP paketa (engl. *IP Header Length*) – inicijalno **5**;
- `ip_tos`: tip usluge (engl. *Type of Service*) – inicijalno **0**;
- `ip_len`: ukupna duljina paketa (engl. *Total Length, Size of Datagram*) – `ukoliko argument nije zadan računa se kao` **20** + duljina polja s podacima;
- `ip_id`: identifikacijska oznaka paketa (engl. *Identification*) – ukoliko argument nije zadan, generira se slučajni broj;
- `ip_off`: *Fragment Offset* – inicijalno **0**;
- `ip_ttl`: vrijeme života paketa (engl. *Time to Live*) – inicijalno **64**;
- `ip_p`: protokol (engl. *Protocol*) – inicijalno **0**;
- `ip_sum`: kontrolna suma (engl. *Header Checksum*) – ukoliko nije zadana, računa se automatski;
- `ip_src`: IP adresa izvora (engl. *Source Address*) – zadaje se kao znakovni niz;
- `ip_dst`: IP adresa odredišta (engl. *Destination Address*) – ovaj argument se može zadati, no prevodilac ga ignorira, budući da se paketi uvijek šalju na ciljnu dresu računala koje se testira.

Funkcija prima imenovane argumente. Argumentom `data` se predaje sadržaj paketa, dok preostali argumenti služe za postavljanje vrijednosti pojedinih polja u zaglavlju IP paketa. Većini argumenata nije potrebno postaviti vrijednost, jer je ona inicijalno postavljena ili se automatski izračunava prilikom kreiranja paketa. Posebno treba naglasiti kako funkcija ignorira argument `ip_dst` ukoliko je on uopće zadan – odredište paketa uvijek je testirano računalo.

Za dohvat pojedinih polja iz IP paketa koristi se funkcija `get_ip_element()`. Funkcija prima dva imenovana argumenta – ime polja koje treba dohvatiti i IP paket iz kojeg se podatak dohvaća. U primjeru je prikazan dohvat *Internet Header Length* elementa iz IP paketa pohranjenog u varijabli `ip_packet`:

```
hl = get_ip_element(ip: ip_packet, element: "ip_hl");
```

Vrijednost pojedinog polja unutar postojećeg IP paketa može se zadati pomoću funkcije `set_ip_element()`. Prvi argument funkcije je, kao i kod prethodne funkcije, varijabla koja je povezana s IP paketom. Ostali argumenti funkcije imaju jednake nazive i značenja kao i argumenti funkcije `forge_ip_packet()`. Bitno je naglasiti da funkcija ne mijenja originalni IP paket, već vraća novi objekt.

4.3.3. Rukovanje TCP i UDP paketima

Za kreiranje TCP paketa koristi se funkcija `forge_tcp_packet()`. Ova funkcija vraća modificirani IP paket u koji je enkapsuliran TCP paket (važno je naglasiti da funkcija ne mijenja `ip_p` polje u IP paketu). Argumenti funkcije su:

- `th_sport`: izvorišni port (engl. *Source Port*);
- `th_dport`: odredišni port (engl. *Destination Port*);

- th_seq: sekvencijalni broj (engl. *Sequence Number*);
- th_ack: potvrdni broj (engl. *Acknowledgement Number*);
- th_off: veličina zaglavlja u 32-bitnim riječima (engl. *Data Offset*), inicijalno **5**;
- th_x2: rezervirano polje (engl. *Reserved*), vrijednost ovog polja ne preporuča se mijenjati, osim ukoliko to ne zahtjeva specifičnost testa;
- th_flags: TCP zastavice (engl. *TCP Flags*);
- th_win: veličina TCP prozora (engl. *TCP Window Size*), inicijalno **0**;
- th_sum: kontrolna suma (engl. *Checkum*) – ukoliko nije zadana, računa se automatski;
- th_urp: zastavica *urgent* (engl. *Urgent Pointer*), inicijalno **0**;
- ip: objekt koji sadrži IP paket u koji treba enkapsulirati stvoreni TCP paket;
- data: podaci (engl. *Data*);
- update_ip_len: zastavica koja kazuje NASL-u da li je potrebno ponovno izračunati duljinu IP paketa (u objektu ip), inicijalno TRUE.

Za kreiranje UDP paketa koristi se funkcija forge_udp_packet(). Kao i prethodna, ova funkcija vraća modificirani IP paket u koji je enkapsuliran UDP paket (važno je naglasiti da funkcija ne mijenja ip_p polje u IP paketu). Argumenti ove funkcije su:

- uh_sport: izvorišni port (engl. *Source Port*);
- uh_dport: odredišni port (engl. *Destination Port*);
- uh_len: duljina UDP pakete (engl . *Length*) – ukoliko argument nije zadan računa se kao duljina zaglavlja + duljina polja data;
- uh_sum: kontrolna suma (engl. *Checkum*) – ukoliko nije zadana, računa se automatski;
- ip: objekt koji sadrži IP paket u koji treba enkapsulirati stvoreni UDP paket;
- data: podaci (engl. *Data*);
- update_ip_len: zastavica koja kazuje NASL-u da li je potrebno ponovno izračunati duljinu IP paketa (u objektu ip), inicijalno TRUE.

Kao i u slučaju IP paketa, postoje i funkcije za čitanje i izmjenu već stvorenih paketa. Za dohvat polja iz paketa služe funkcije get_tcp_element() i get_udp_element(). Funkcije primaju iste parametre kao i ranije spomenuta funkcija get_ip_element(). Za izmjenu polja u paketu predviđene su funkcije set_tcp_element() i set_udp_element(). Ove funkcije rade slično kao i funkcija set_ip_element ().

4.3.4. Rukovanje ICMP i IGMP paketima

Osim TCP i UDP paketa, u IP pakete moguće je enkapsulirati i ICMP, odnosno IGMP protokole. Međutim, funkcije koje služe za rukovanje takvim paketima (forge_icmp_packet, forge_igmp_packet i get_icmp_element) nešto su manjih mogućnosti od funkcija za rukovanje TCP, UDP i IP paketima. Na primjer, programer pomoću funkcije forge_icmp_packet ne može definirati sva polja unutar ICMP zaglavlja, već samo ona osnovna. Argumenti funkcije navedeni su u nastavku:

- icmp_cksum: kontrolna suma (engl. *Checkum*);
- icmp_code: ICMP kod paketa (engl. *ICMP code*);
- icmp_id: identifikacijska oznaka paketa;
- icmp_seq: sekvencijalni broj paketa (engl. *ICMP sequence number*);
- icmp_type: tip ICMP paketa (engl. *ICMP type*);
- ip: objekt koji sadrži IP paket u koji treba enkapsulirati stvoreni ICMP paket;
- data: podaci (engl. *Data*);

- `update_ip_len`: zastavica koja kazuje NASL-u da li je potrebno ponovno izračunati duljinu IP paketa (u objektu `ip`), inicijalno TRUE.

4.3.5. Slanje i primanje ručno oblikovanih paketa

Za slanje ručno oblikovanih paketa (paketa oblikovanih pomoću neke od `forge_*_packet()` funkcija) i primanje odgovora koristi se funkcija `send_packet()` čiji su argumenti:

- `length`: duljina paketa;
- `pcap_active`: argument koji određuje da li će NASL čekati odgovor na paket, inicijalno TRUE;
- `pcap_filter`: BPF filter (znakovni niz koji određuje koji se primljeni odgovori uzimaju u obzir), inicijalna vrijednost je "**ip and (src host `target` and dst host `nessus_host`)**";
- `pcap_timeout`: vrijeme nakon kojeg funkcija prestaje čekati odgovor, inicijalno **5** sekundi.

Sljedeći odlomak koda prikazuje primjer kreiranja ICMP *echo* paketa i čekanja odgovora na isti:

```
id = rand() % 65534;

ip_packet = forge_ip_packet(
ip_v: 4,
ip_hl: 5,
ip_tos: 6,
ip_len: 20,
ip_id: 0x4747,
ip_off: IP_DF,
ip_ttl: 0x40,
ip_p: IPPROTO_ICMP,
ip_src: this_host()
);

icmp_packet = forge_icmp_packet(
ip: ip_packet,
icmp_type: 13,
icmp_code: 123,
icmp_seq: id,
icmp_id: id);

filter = "icmp and src host " + get host ip() + "
andicmp[0:1]=0 and icmp[6:2]=" + id;

reply = send packet(icmp packet, pcap active:TRUE,
pcap_filter: filter, pcap_timeout: 1);
```

4.4. Funkcije za komunikaciju putem TCP i UDP protokola

Za jednostavniju komunikaciju putem TCP i UDP protokola implementirane su specijalno prilagođene funkcije koje omogućavaju jednostavno kreiranje veze te razmjenu podataka. Ove funkcije vrlo su slične onima u C programskom jeziku te su iz tog razloga ovdje samo ukratko opisane. Detaljne informacije moguće naći u dokumentaciji koja je navedena na kraju dokumenta (Poglavlje 8).

4.4.1. Otvaranje veze

Za otvaranje TCP i UDP mrežnih utičnica (engl. *socket*) koriste se funkcije `open_sock_tcp` i `open_sock_udp`. Obje funkcije primaju jedan neimenovani parametar – broj mrežnog porta na udaljenom računalu. Također, postoje i funkcije `open_priv_sock_tcp` te `open_priv_sock_udp`, koje služe za otvaranje privilegirane utičnice. Ove dvije funkcije primaju imenovane parametre koji zadaju izvorišni i odredišni mrežni port (parametri `sport` i `dport`).

4.4.2. Slanje i primanje podataka

Nakon otvaranja mrežne utičnice, putem nje je moguća razmjena podataka sa servisom na udaljenom računalu.

Za slanje podataka koristi se funkcija `send()`. Ova funkcija prima sljedeće imenovane parametre:

- `socket`: objekt koji predstavlja otvorenu utičnicu;
- `data`: blok podataka ("čisti" ili "nečisti" znakovni niz);
- `length`: ukupna duljina podataka, inicijalno se postavlja na duljinu znakovnog niza predanog putem `data` argumenta;
- `option`: zastavice koje se proslijeđuju sistemskom pozivu `send()`.

Za primanje podataka koriste se funkcije `recv()` i `recv_line()`. Obje funkcije vraćaju znakovni niz s pročitanim podacima, a primaju sljedeće imenovane parametre:

- `socket`: objekt koji predstavlja otvorenu utičnicu;
- `length`: ukupna duljina podataka koje treba pročitati.

Razlika između funkcija `recv()` i `recv_line()` je u tome što funkcija `recv_line()` staje s čitanjem kada pročita znak za kraj linije (engl. *line feed*).

4.5. Uključivanje funkcija iz biblioteke funkcija

Osim ugrađenih funkcija opisanih u prethodnim poglavljima, uz NASL inerpreter dolazi i specijalna biblioteka, kojoj je osnovna namjena da programerima olakša izvođenje postupaka koji su česti prilikom provođenja postupaka ispitivanja sigurnosti. Primjeri takvih postupaka su provjera inačice servisa dohvatom tzv. *bannera*, pristupanje dijeljenim direktorijima na udaljenom računalu putem NFS ili Samba protokola, slanje HTTP zahtjeva i primanje odgovora i sl. Kako se radi o biblioteci funkcija koja je još u razvoju, broj funkcija u biblioteci je relativno skroman, a njihove su mogućnosti za sada prilično ograničene.

Biblioteke funkcija koje su uključene u NASL paket nalaze se u datotekama s nastavkom `.inc`, a instalacijski će ih program po inicijalnim postavkama raspakirati u direktorij `/var/lib/nessus/plugins`. Za uključivanje pojedinih biblioteka funkcija koristi se funkcija `include()`:

```
include("/var/lib/nessus/plugins/http_func.inc");
```

Detaljan popis funkcija i njihovih parametara moguće je naći u dokumentaciji navedenoj na kraju dokumenta

5. Pisanje Nessus modula za ispitivanje ranjivosti

Prilikom pisanja modula za provjeru ranjivosti potrebno je obratiti pozornost na nekoliko ključnih elemenata. Prvo, svaka Nessus kripta prilikom pokretanja, Nessus poslužitelju mora proslijediti odgovarajuće podatke, ili je on uopće neće izvršiti. Struktura Nessus skripte detaljnije je opisana u poglavlju 4.1, a dio skripte zadužen za dostavljanje spomenutih podataka Nessus poslužitelju u poglavlju 4.2. Prilikom izrade skripti potrebno je voditi računa o ovim detaljima, budući da se u suprotnom one neće moći izvršiti.

Nadalje, nakon što skripta utvrdi da je određena ranjivost prisutna na testiranim sustavu, istu je korisniku potrebno prijaviti u konačnom izvještaju o obavljenom ispitivanju. Funkcije koje služe za prijavljivanje sigurnosnih propusta opisane su u poglavlju 4.3. Također, vrlo je važno da skripte budu programirane tako da se test izvrši u što kraćem vremenskom periodu. O efikasnosti skripti i metodama njihovog unapređenija više je riječi dano u poglavlju 4.4. Konačno, ukoliko skripta nije namijenjena samo za internu uporabu, nužno je prilikom njene izrade poštovati određena pravila koja su detaljnije opisana u poglavlju 4.5.

5.1. Struktura Nessus skripte

NASL skripte pisane za Nessus alat moraju se sastojati od dva dijela:

- **registracija** – dio skripte koji Nessus poslužitelju daje podatke o autoru, tipu, namjeni skripte i sl.;
- **testiranje** – dio skripte koji provjerava postoji li na testiranom sustavu tražena ranjivost.

Struktura takve skripte izgleda ovako:

```
if(description)
{
#
# REGISTRACIJA
#
exit(0);
}
#
# TESTIRANJE
#
```

Vrijednost `description` je ugrađena konstanta koju postavlja Nessus poslužitelj. Prilikom prvog parsiranja skripte njena će vrijednost biti **1**, te će se izvršiti samo registracija (radi poziva funkcije `exit` na kraju registracijskog dijela). Prilikom svakog sljedećeg pokretanja njena će vrijednost biti **0**, te će se registracija preskočiti, a izvršiti samo onaj dio koji ispituje ranjivost na udaljenom sustavu.

5.2. Registriranje skripte

Dio zadužen za registraciju **mora** pozivati sljedeće funkcije:

- `script_id(<id>)` – postavlja identifikacijsku oznaku skripte;
- `script_version(<version>)` – postavlja oznaku inačice skripte;
- `script_name(english:<name>[, ...])` – postavlja ime skripte koje će se prikazati u Nessus klijentu;
- `script_description(english:<desc>[, ...])` – postavlja detaljniji opis skripte koji korisnicima daje uvid u njenu namjenu i način rada;
- `script_summary(english:<summary>[, ...])` – postavlja skraćeni opis skripte koji mora stati u jedan redak;
- `script_category(<category>)` – postavlja kategoriju skripte, moguće vrijednosti argumenta su:
    1. `ACT_GATHER_INFO` – skripta koja samo prikuplja informacije, pri čemu pouzdano ne može naštetiti udaljenom računalu;
    2. `ACT_SCANNER` –skripta za pregledavanje mrežnih portova (engl. *port scanner*);
    3. `ACT_ATTACK` – skripta koja prilikom ispitivanja pokušava doći do ovlasti na udaljenom računalu i pri tome mu mogu naštetiti;

4. ACT_DENIAL – skripta koja prilikom ispitivanja provodi napad koji može uzrokovati prekid rada udaljenog računala ili servisa na njemu (engl. *Denial of Service*,DoS).

- `script_copyright(english:<copyright>[, ...])` – postavlja informaciju o autorstvu skripte (najčešće ime autora);
- `script_family(english:<family>[, ...])` – postavlja tip skripte.

Identifikacijska oznaka skripte jest jedinstveni broj po kojem Nessus poslužitelj razlikuje različite module za ispitivanje ranjivosti. Skriptama koje se distribuiraju zajedno sa Nessus programskim paketom identifikacijske oznake dodjeljuju ovlašteni članovi Nessus projekta (www.nessus.org), dok skriptama za internu upotrebu identifikacijske oznake pridjeljuju korisnici prema vlastitom odabiru (oznake između **90000** i **99000)**. Inačica, ime, skraćeni i detaljni opis skripte te informacije o autorstvu podaci su koji se pojavljuju u korisničkom sučelju klijentske aplikacije.

Neki od mogućih tipova skripti su: *Backdoors*, *Denial of Service*, *Firewalls*, *Gain a shell remotely*, *Gain root remotely*, *Remote file access* i sl., no korisnik može postaviti tip skripte na proizvoljnu vrijednost. Ipak, ukoliko se radi o skripti namijenjenoj slobodnoj distribuciji, preporuča se navođenje jednog od uobičajenih tipova (detaljni popis podržanih tipova moguće je naći na adresi http://www.nessus.org/plugins/).

Mnoge od navedenih funkcija imaju argumente imenovane prema jezicima. Mogući imenovani argumenti su `english`, `french`, `deutsch` i `portuguese`. Moguće je i umjesto imenovanih argumenata navesti jedan neimenovani, pri čemu se pretpostavlja da je riječ o engleskom jeziku.

Osim navedenih funkcija, u ovom dijelu skripte mogu se pozvati i mnoge druge funkcije koje postavljaju različite parametre modula. Neke od zanimljivijih su:

- `script_dependencies(<filename>[, ...])` – daje Nessus programu do znanja da ova skripta koristi rezultate drugih modula, te da mora biti pokrenuti **nakon** njih;
- `script_require_ports` – test se provodi samo ukoliko su navedeni mrežni portovi otvoreni na udaljenom računalu:

```
script_require_ports (23, "Services/telnet);
```

- `script_timeout` – postavlja vrijeme nakon kojeg se rad skripte prekida (vrijednosti **0** i **-1** znače da je vrijeme neograničeno);
- `script_cve_id` – CVE identifikacijski broj ranjivosti (http://cve.mitre.org);
- `script_bugtraq_id` – *Bugtraq* identifikacijski broj propusta koji skripta traži na udaljenom računalu.

Cjeloviti popis funkcija i njihovih parametara moguće je naći u dokumentaciji NASL navedenoj na kraju dokumenta (Poglavlje 8).

5.3. Prijavljivanje sigurnosnih propusta

Za prijavljivanje identificiranih sigurnosnih propusta koriste se funkcije `security_warning()`, `security_hole()` i `security_note()`.

Funkcija `security_note()` koristi se ukoliko je sigurnosni propust utvrđen na udaljenom računalu, no ne predstavlja izravnu prijetnju za testirano računalo. Tipično, to su propusti koji napadaču otkrivaju informacije koje mogu pomoći prilikom planiranja napada – informacije o pokrenutim servisima, lokaciji podataka na tvrdom disku računala i sl. Funkcija `security_warning()` koristi se ukoliko je sigurnosni propust utvrđen, no ne predstavlja ozbiljnu prijetnju – njegovim iskorištavanjem napadač ne može preuzeti potpunu kontrolu nad udaljenim računalom. Funkcija `security_hole()` koristi se ukoliko je sigurnosni propust utvrđen i predstavlja ozbiljnu sigurnosnu prijetnju – njegovim iskorištavanjem napadač najčešće može doći do ovlasti na udaljenom računalu.

Parametri ovih funkcija su:

`data` – tekst koji se pojavljuje u izvješću (ukoliko je ovaj argument izostavljen, u izvješću će biti detaljniji opis postavljen pomoću `script_description` funkcije);

`port` – mrežni port na kojem je propust utvrđen (ovaj argument se izostavlja ukoliko se propust odnosi na sve servise);

`proto` (ili `protocol`) – mrežni protokol (inicijalno `"tcp"`, druga moguća vrijednost je `"udp"`).

5.4. Efikasnost skripti

Prilikom programiranja Nessus modula vrlo je važno obratiti pozornost na brzinu rada skripti. Brzina izvođenja skripti posebno dolazi do izražaja kada se istovremeno testira veći broj računala (što je najčešće i slučaj), budući da se u tom slučaju vrijeme testiranja može znatno produljiti. Osnovno pravilo koje je potrebno poštivati s ciljem povećanja efikasnosti je da skripta ne smiju raditi ništa što **nije moguće** ili je **već ranije učinjeno**. U nastavku su navedene neke od funkcija koje mogu pomoći u ovom pogledu.

5.4.1. Funkcija `get_port_state()`

Ova funkcija provjerava da li je mrežni port na udaljenom računalu otvoren. Ukoliko je port otvoren, vreća vrijednost `TRUE`, a u suprotnom `FALSE`. Poziv ove funkcije oduzima vrlo malo procesorskog vremena, a njime se može izbjeći dugotrajno čekanje na odgovor (recimo u slučaju pokušaja spajanja na port zaštićen vatrozidom). Valja napomenuti da u slučaju kada je stanje mrežnog porta nepoznato (nije provjereno u do tog trenutka provedenim testovima), funkcija također vraća vrijednost `TRUE`.

Sintaksa poziva je vrlo jednostavna i opisana je u nastavku,

```
get_port_state(<port>);
```

gdje je `<port>` broj mrežnog porta koji se ispituje.

5.4.2. Baza znanja

Baza znanja (engl. *Knowledge base*) je interna baza podataka u kojoj se pohranjuju različiti podaci prikupljeni tijekom ispitivanja udaljenog računala, pri čemu svako testirano računalo dobiva svoju vlastitu bazu. Skripte putem ove baze razmjenjuju podatke koji im mogu pomoći da uspješno i u što kraćem vremenu izvrše testiranje – aktivne mrežne portove, pokrenute servise, podatke o konfiguraciji pojedinih servisa, inačicu operacijskog sustava i sl.

Ključ se sastoji od tri podatka – kategorije, imena i vrijednosti. Kategorija i ime se zadaju u istome znakovnom nizu, razdvojeni znakom `/`. Primjer takvog znakovnog niza je `"Services/www"`. Kategorija *Services* objedinjava ključeve čija vrijednost označava mrežni port na kojem je aktivan određeni mrežni servis (u ovom slučaju to je *Worl Wide Web* servis) Prethodni ključ tako će tipično imati vrijednost **80**. Ukoliko je na poslužitelju paralelno pokrenut još jedan WWW servis, u bazu znanja upisati će se još jedan ključ s istim imenom (i kategorijom, naravno), ali drugom vrijednošću. Valja napomenuti da kategorija **nije** obavezan dio ključa te se prilikom upisa novog ključa u bazu znanja može izostaviti.

Više o elementima baze znanja moguće je naći u dokumentaciji o NASL jeziku navedenoj na kraju dokumenta (Poglavlje 8).

Za dohvat, čitanje, postavljanje i ažuriranje ključeva u bazi znanja koriste se sljedeće funkcije:

- `set_kb_item(name: <name>, value: <value>)` – služi za postavljanje novog ključa u bazu znanja. Prima dva imenovana argumenta, ime ključa i vrijednost. Primjerice, želimo li postaviti vrijednost ključa iz gore opisanog primjera, poziv funkcije izgledati će ovako:

```
set_kb_item(name: "Services/www", value: 80);
```

  Uzastopni pozivi ove funkcije s istim imenom ključa u bazu unose više vrijednosti ključa (stvara se lista vrijednosti)!

- `get_kb_item(<name>)` – čita vrijednost ključa čije je ime zadano kao neimenovani argument. Primjer poziva dan je u nastavku:

```
if (get_kb_item(name: "Services/www") == 80)
{
  # ovaj odsje•ak izvrišiti •e se samo ukoliko je vrijednost
  # klju•a "Services/www" postavljena na 80
}
```

- `get_kb_list(<mask>)` – dohvaća više ključeva iz baze znanja. Neimenovani argument `<mask>` je ime ključa, u kojem određeni dio može biti zamijenjen znakom `*`. Sljedeći primjer ilustrira upotrebu ove funkcije:

```
get_kb_list("Services/www");  # svi klju•evi koji
            # ozna•avaju portove s aktivnim web servisima
get_kb_list("Services/*");          # svi klju•evi iz
            # kategorije "services", tj. svi servisi
get_kb_list("*");             # svi klju•evi iz baze
```

  Funkcija vraća vrijednost koja se može iterirati korištenjem programske petlje `foreach`, ili pretvoriti u listu korištenjem funkcije `make_list()`.

- `replace_kb_item(name: <name>, value: <value>)` – mijenja vrijednost ključa u bazi, ili dodaje novu vrijednost (ukoliko ključ s imenom već ne postoji u bazi). Uzastopni pozivi ove funkcije neće stvoriti listu, kao što je to slučaj s funkcijom `set_kb_item()`.

5.4.3. Funkcije za uvjetovanje izvršavanja skripte

Tijekom postupka provjere ranjivosti često se pojavljuju situacije u kojima nije potrebno provoditi ispitivanje određenog servisa budući da je on ili neaktivan, ili je podešen tako da ranjivost nije moguće iskoristiti i sl.

Sljedeće funkcije koje se navode u registracijskom dijelu skripte postavljaju uvjete za pokretanje određenog testa:

- `script_require_ports(<port1>[, <port2>[, ...]])` – provjerava da li je port (ili više njih) otvoren. Ukoliko nije, Nessus poslužitelj neće pokrenuti izvođenje skripte. `<port>` u ovom slučaju može biti cijeli broj, ili ključ iz baze znanja koji se odnosi na određeni servis. Primjer poziva funkcije dan je u nastavku:

```
script_require_ports(80, ""Services/www");
```

- `script_require_keys(<key1>[, <key2>[, ...]])` – provjerava da li je zadani ključ (ili više njih) sadržan u bazi znanja. Ukoliko neki od navedenih ključeva ne postoji u bazi znanja, izvođenje skripte se neće pokrenuti.
- `script_exclude keys(<key1>[, <key2>[, ...]])` – radi obrnuto od prethodne funkcije. Izvođenje se neće pokrenuti ukoliko neki od navedenih ključeva **postoji** u bazi znanja.

5.5. Pravila za oblikovanje skripti

Prilikom izrade skripti za osobnu upotrebu korisnik se ne mora pridržavati nikakvih pravila osim, naravno, onih koje nameće sama gramatika jezika. No, ukoliko je riječ o modulu koji je pisan s namjerom javnog objavljivanja i daljnje distribucije te eventualnog uključivanja Nessus programski paket, postoje pravila kojih se programer mora pridržavati:

- **između modula i korisnika nema interakcije** – kako se modul izvršava na poslužiteljskoj strani, jedini način dojavljivanja rezultata korisniku je putem funkcija opisanih u poglavlju 5.3.;

- **svaki modul provjerava jednu ranjivost** – radi konzistencije s ostalim modulima, od modula se očekuje da provjeravaju točno jednu ranjivost. Ukoliko je riječ o povezanim ranjivostima, moduli mogu komunicirati putem baze znanja i zavisnosti;
- **poželjno je koristiti postojeći tip skripte (engl. *script family*)** – ne preporuča se korištenje novih tipova ranjivosti;
- **pridržavanje CVE standarda** – ukoliko skripta ispituje ranjivost koja je već opisana u CVE bazi ranjivosti, modul bi trebao sadržavati informaciju o tome;
- **identifikacijsku oznaku modula dodjeljuju ovlaštene osobe** – o uključivanju modula koji ispituju novootkrivene ranjivosti odlučuju ovlaštene osobe, koje pritom modulu dodjeljuju jedinstvenu identifikacijsku oznaku (*ID*).

6. Primjer

U nastavku je dan jednostavan primjer potpuno funkcionalnog modula za Nessus. Zadaća je modula da pokuša utvrditi inačicu Web poslužitelja slanjem jednostavnog HTTP upita oblika `"HEAD / HTTP/1.0\r\n\r\n"`.

Svrha ovog primjera je isključivo demonstracija programiranja u NASL jeziku. U distribuciji Nessus programskog paketa već postoje mnogo složeniji moduli za utvrđivanje inačica Web poslužitelja te se u praksi preporučuje njihovo korištenje.

```
if(description)
{
  script_id(99001);
  script_version("$Revision 1.0$");
  script_name(english:"My HTTP script");
  script_description(english:"This  script  sends  a  trivial
                     request  to  the  Web  server  and  tries  to
                     figure out it's version.");
  script_summary(english:"Checks the Web server version");
  script_category(ACT_GATHER_INFO);
  script_family(english:"My Private Scripts");
  script_copyright(english:"Script written by ME");
  script_dependencies("find_service.nes");
  exit(0);
}


include("/var/lib/nessus/plugins/http_func.inc");

  http_port = get_http_port(default: 80);

  if (get_port_state(http_port))
  {
    http_socket = http_open_socket(http_port);
    if (http_socket)
    {
      request = string("HEAD / HTTP/1.0\r\n\r\n");
      send (socket: http_socket, data: request);
      reply = http_recv_headers (http_socket);
      http_close_socket (http_socket);
```

```
      server version = egrep(string: reply, pattern:"^Server:
                       *[^ \t\n\r]")
      if (strlen(server_version))
      {
        report   =   "Remote   HTTP   server   leaks   version
                      information about it'sversion:

"       + server_version;
        security_note (port: http_port, data: report);
        display (server_version, "\n\n");
      }
    }
  }

exit(0);
```

7. Zaključak

Iz dokumenta je moguće zaključiti kako NASL (engl. *Nessus Attack Scripting Language*) predstavlja vrlo jednostavan, a opet moćan alat za izradu modula za ispitivanje sigurnosnih propusta u računalnim sustavima. Jezik je vrlo jednostavan, a većina njegovih pravila bazira se na C programskom jeziku što programerima dodatno olakšava njegovo korištenje. Korištenjem NASL jezika sigurnosni stručnjaci i sistem administratori mogu u vrlo kratkom vremenu kreirati vlastite skripte koje im mogu olakšati u redovitom održavanju sustava te uočavanju potencijalnih sigurnosnih propusta. Također, otvorenost jezika pridonosi bržem razvoju modula za ispitivanje sigurnosnih problema korištenjem Nessus programa, što je također jedna od njegovih kvaliteta.

8. Reference

[1] The NASL2 reference manual - http://michel.arboi.free.fr/nasl2ref/nasl2_reference.html

[2] The Nessus Attack Scripting Language Reference Guide, http://www.virtualblueness.net/nasl.html

[3] NASL Coding Tutorial - http://www.secguru.com/forum/viewtopic.php?t=121&postdays=0&postorder=asc&start=0

[4] Nessus, http://www.nessus.org/