MRLAB

Stack əsaslı bufer daşması və uzaqdan kod icrası

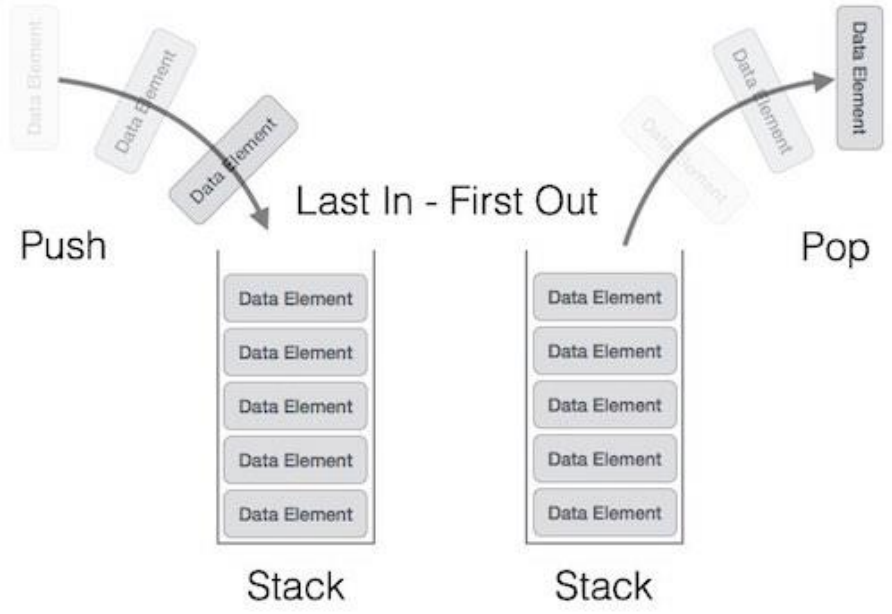
Azərbaycan Respublikası Xüsusi Rabitə və İnformasiya Təhlükəsizliyi Dövlət Xidməti – Kompüter İnsidentlərinə qarşı Mübarizə Mərkəzi -Malware Research Lab – S. Abasov - 12 May 2022

Bu məqalədə sizlərə komputer sistemlərinə uzaqdan müdaxilə üçün istifadə edilən exploit növlərindən biri olan buffer overflow (yaddaş daşması) haqqında danışacağam.

BOF - proqramçının yazdığı proqram təminatında yaddaş daşmasının qarşısını almaq üçün lazımı yoxlamaları etmədiyi üçün baş verən bir xətadır. Lakin bu səhv olduqca kritik səhv olduğu üçün pis məqsədlər üçün istifadə edilməyə başlanılıb. Bir çox növü olmasına baxmayaraq bu məqalədə stack overflow metodundan və bu boşluqdan faydalanaraq qarşı sistemə necə müdaxilə etmək olar bu haqda bəhs edəcəyəm.

Stack nədir?

Windows əməliyyat sistemində stack yaddaş bölgəsi proqram təminatlarının fəaliyyəti üçün olduqca kritik bölgədir. Proqram təminatına aid bir çox önəmli məlumatlar məhz bu bölgədə saxlanılır. Stack data strukturu LIFO(Last in First Out) anlayışı ilə işləyir. Yəni *son daxil olan məlumat ilk çıxır*. Bunu daha yaxşı təsəvvür etmək üçün boşqab misalına baxaq. Yuyulan boşqablar quruması üçün ayrı bir yerə yığılır daha sonra həmin boşqablar quruduqdan sonra çəkməcəyə qoyulur. Burada yuyulan son boşqab quruduqdan sonra çəkməcəyə qoyulacaq ilk boşqabdır. Stackin işləmə prinsipidə eyni ilə belədir. *Daxil olan ilk məlumat son, daxil olan son məlumat isə ilk çıxır*.



Stack bölgəsində xüsusi ilə lokal dəyişənlər, funksiya geri dönüş adresləri vs kimi məlumatlar saxlanılır. Bunu daha yaxşı izah etmək üçün pratik olaraq bir test keçirək və hazırladığımız kiçik proqram təminatının stack bölgəsində hansı məlumatları saxladığına baxaq. Bunun üçün debugger alətindən istifadə ediləcəkdir.Kompilyatorun kod optimizasiya mexanizmi istifadə edilməyəcək.

```
DWORD B(DWORD P1, DWORD P2)
{
	DWORD dwNetice2 = 0;
	dwNetice2 = P1 + P2;

	return dwNetice2;
}

DWORD A(DWORD P1, DWORD P2)
{
	DWORD dwNetice1 = 0;
	dwNetice1 = B(P1, P2);

	return dwNetice1;
}

int main(void)
{
	A(1, 2);
}
```

Burada ilk olaraq main funksiyasına baxaq.

```
push ebp
mov ebp,esp
push 2
push 1
call <exploit._A>
add esp,8
xor eax,eax
pop ebp
ret
```

Burada ilk olaraq push ebp, mov ebp, esp əməliyyatları ilə esp – yəni stack pointeri (stack bölgəsini) ebp registeri üzərindən idarə etmək üçün kiçik bir əməliyyat icra edilir. Daha sonra push 2 və push 1 instructionları ilə 1 və 2 rəqəmlərini stack bölgəsində yazır. Bu rəqəmlər çağrılacaq A funksiyasının parametrləridir. Prametrlər yazıldıqdan sonra isə A funksiyası çağrılır. A funksiyası çağrılmazdan öncə stack bölgəsinə baxaq.

```
00DEFCDC 00000001
00DEFCE0 00000002
```

```
00DEFCD8 008A1C9C return to exploit.main+C from exploit._A
00DEFCDC 00000001
00DEFCE0 00000002
00DEFCE4 00DEFD2C
```

Burada diqqət etməli olduğunuz məqam 0x008A1C9C dəyəridir. Bunu stack bölgəsinə yazan call instructionudur. Main funksiyası A funksiyasını çağırdıqdan sonra A funksiyası özü işini görür və daha sonra yenidən main funksiyasına qayıtmalıdır. Hara qayıdacağını isə bu stack bölgəsinə call instr ilə yazılan bu dəyər təyin edir. Ret instruction funksiyas öz işlərini bitirdikdən sonra stackdən geri dönəcəyi adresi alır və həmin adresə qayıdır. Bu adresi isə main funksiyasından call əməliyyatından sonra ki, adresi göstərir.

```
008A1C90 <exploit._mai $  55          push ebp
008A1C91         .  8BEC        mov ebp,esp
008A1C93         .  6A 02       push 2
008A1C95         .  6A 01       push 1
008A1C97         .  E8 14000000 call <exploit._A>
008A1C9C         .  83C4 08     add esp,8
```

Daha sonra B funksiyası çağrılır. Tərkib koduna baxsanız A funksiyasının öz növbəsində main funksiyasından gələn parametrləri B funksiyasına toplama əməliyyatı üçün göndərir və gələn nəticəni dwNetice1 adlı lokal dəyişəndə saxlayır. Gəlin A funksiyasının bunu necə etdiyinə baxaq.

```
push ebp
mov ebp,esp
push ecx
mov dword ptr ss:[ebp-4],0
mov eax,dword ptr ss:[ebp+C]
push eax
mov ecx,dword ptr ss:[ebp+8]
push ecx
call <exploit._B>
```

Burada ilk olaraq mov dword ptr ss:[ebp-4],0 ilə dwNetice1 dəyişəni üçün stack bölgəsində boş yer ayırır və 0 dəyərini yazır. Bunun c dilində qarşılığı - dwNetice1 = 0. Daha sonra main funksiyasından gələn parametrləri (1 və 2) stack bölgəsindən götürür. Götürülən parametrlər push əməliyyatı ilə növbəti (B) funksiyaya göndərilmək üçün yenidən stack bölgəsində yazılır.

```
mov  eax,dword ptr ss:[ebp+C]
push eax
mov  ecx,dword ptr ss:[ebp+8]
push ecx
call <exploit._B>
```

```
005EFED4  00000001
005EFED8  00000002
005EFEDC  00000000
005EFEE0  005EFEF0
005EFEE4  008A1C9C  return to exploit.main+C from exploit._A
005EFEE8  00000001
005EFEEC  00000002
005EFEF0  005EFF38
```

Əməliyyatından öncə stack bölgəsi

```
005EFED0  008A1CC8  return to exploit.A+18 from exploit._B
005EFED4  00000001
005EFED8  00000002
005EFEDC  00000000
005EFEE0  005EFEF0
005EFEE4  008A1C9C  return to exploit.main+C from exploit._A
005EFEE8  00000001
005EFEEC  00000002
005EFEF0  005EFF38
```

Əməliyyatdan sonra stack bölgəsi

Stack Frame

Yuxarıda ki, şəkilə diqqət yetirsəniz debuggerin qara xətlər ilə stack bölgəsini ayırdığını görə bilərsiniz. Bunun səbəbi *stack frame* anlayışıdır. Hər bir prosedurun(funksiyanın) öz stack frame-i olur. Yəni main funksiyasının öz , A funksiyasının öz stack frame-i olur. Funksiyaya aid lokal dəyişənlər geri dönüş adresləri məhz bu frame içərisində saxlanılır. Stack frame funksiya çağrıldığı zaman cari ESP-də yaradılır. Stack haqqında ümumi məlumatlandıqdan sonra keçirik explotasiya qisminə.

Stack Bufferinin explotasiya edilməsi

Stack Bufferinin explotasiya edilməsində ki, məqsəd funksiya geri donüş dəyərinə müdaxilə edərək EIP hücüm edən tərəfinən

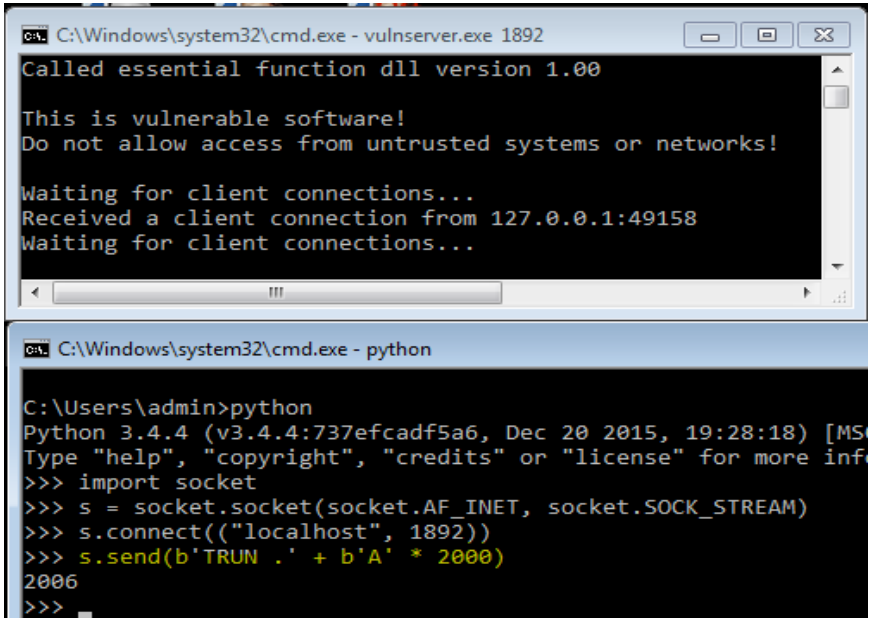
istədiyi kodlara yönləndirməkdir. Bu əksər hallarda elə məhz stack bölgəsinin özü olur. ESP –nin korlanması ilə stack frame içərisində olan funksiya geri dönüş dəyəri lazımı dəyər ilə dəyişdirilir. Bundan sonra idarə etmə tamami ilə explotasiya edən tərəfin əlinə keçir və shell kodu icra edilir. Bunu daha yaxşı anlamaq üçün praktiki olaraq stack overflow boşluğunu explotasiya edərək boşluğun necə istismar olunduğuna baxaq. Test sistemi olaraq Windows 7 x86 English istifadə edəcəyəm. Explotasiya üçün isə stack overflow boşluğu daşıyan Stephen Bradshaw tərəfindən hazırlanan vulnserver applikasiyasından istifadə edəcəyəm. Bu applikasiya məhz bu tip əməliyyatları test etmək üçün hazırlanmışdır. Daha ətraflı: https://thegreycorner.com/vulnserver.html
Vulnserveri normal bir server applikasiya kimi düşünə bilərsiniz. Server sadəcə olaraq qarşıdan gələn məlumatları emal etmək üçün proqramlaşdırılıb. Burada stack overflow boşluğunu aktivləşdirmək üçün TRUN əmrindən istifadə edəcəyik. Tərkib kodlarına baxsaq boşluğun **strcpy** funksiyasında olduğunu görə bilərik. Strcpy funksiyası mətn kopyalama əməliyyatlarından istifadə edilir. Lakin problem funksiyanın daxil olan məlumatın həcmini yoxlamamasındadır.

```
else if (strncmp(RecvBuf, "TRUN ", 5) == 0) {
     char *TrunBuf = malloc(3000);
     memset(TrunBuf, 0, 3000);
     for (i = 5; i < RecvBufLen; i++) {
          if ((char)RecvBuf[i] == '.') {
               strncpy(TrunBuf, RecvBuf, 3000);
               Function3(TrunBuf);
               break;
```

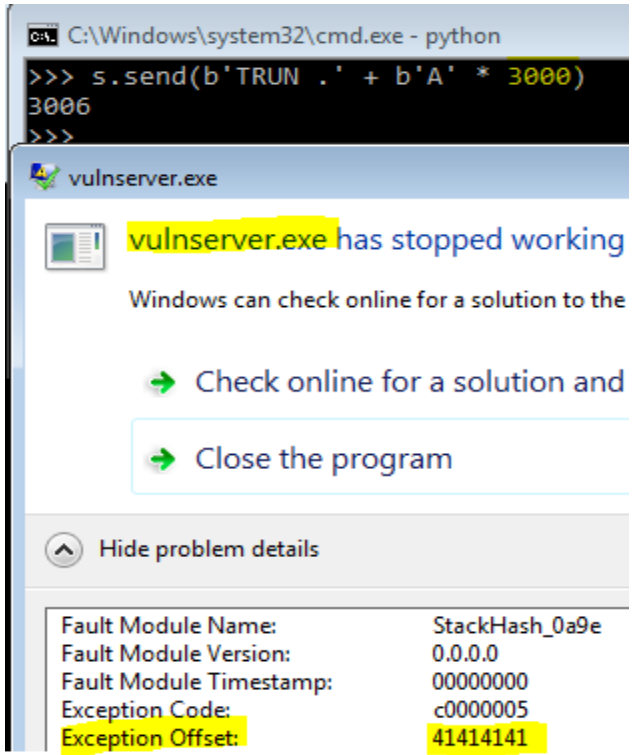
Burada server socket üzərindən gələn datanın **TRUN .** ilə başladığını test etdikdən sonra buradan 3000 baytlıq məlumatı götürərək **Function3** funksiyasına parametr olaraq göndərir. **Function3** isə boşluğun yarandığı funksiyadır.

```
void Function3(char *Input) {
    char Buffer2S[2000];
    strcpy(Buffer2S, Input);
}
```

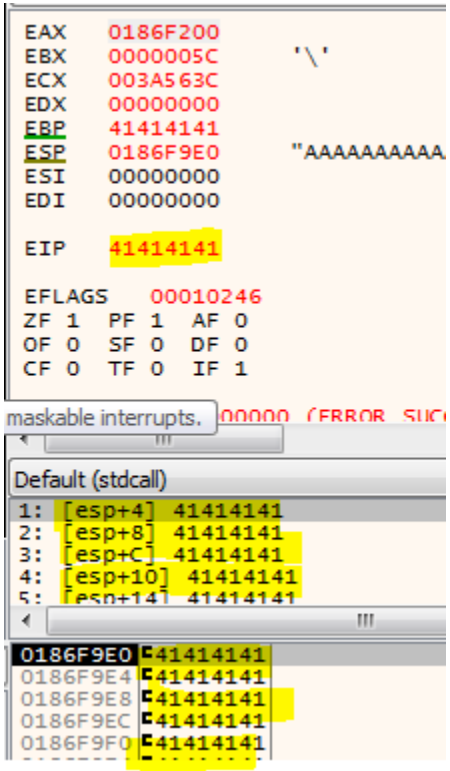
Diqqət etsəniz funksiyası stack framində lokal dəyişən Buffer2S üçün 2000 baytlıq yer ayrılır. Lakin funksiyaya parametr olaraq göndərilən dəyər bundan daha çoxdur. Belə olan halda buffer(stack) daşması baş verir. Test edirik. İlk olaraq **vulnserver 1892** (lokal port) serveri qaldırırıq və python ilə serverə məlumat göndəririk.



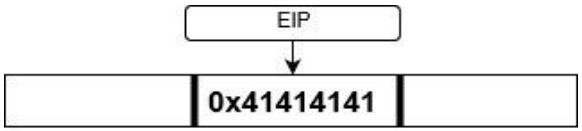
Əmri göndərikdən sonra proqram normal fəaliyətinə davam edir. Gəlin daha çox məlumat göndərərək test edək. Bu dəfə əmr + 3000 ədəd A xarakterini göndərirəm. Windows Error Report vulnserver də baş verən exception haqqında məlumat göstərir. Burada diqqət edilməli məqam **Exception Offsetdir**. Exception baş verdiyi adres **0x41414141** adresidir. Lakin bu normal adres deyil. Xatırlayırsınız sa biz vulnserver-ə "A" xarakterini göndərmiş idik. Bu xarakterin hexa qarşılığı isə 0x41-dir. Deməli EIP registeri korlanaraq bizim göndərdiyimiz A xarakteri ilə əvəzlənib. Bunun səbəbi isə stack framedə saxlanan öncəki funksiyanın geri dönüş adresidir.



Bizim göndərdiyimiz 3000 baytlıq məlumat stack bölgəsini korlayaraq EIP-nin əvəzlənməsinə səbəb olub. Həmçinin stack yaddaşıda eyni şəkildə. Vulnserver-i debug edərək daha ətraflı məlumat görə bilərik.



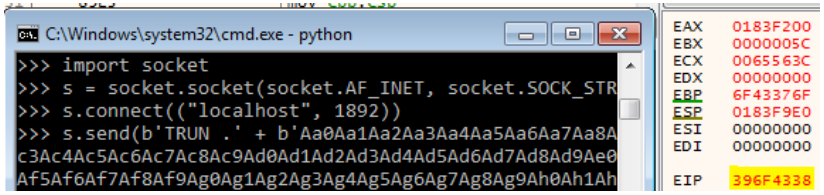
Gördüyünüz kimi stack tamami ilə korlanıb və bu birbaşa EIP registerinə təsir edib. Bundan sonra EIP –ni düzgün şəkildə yönləndirərək shell kodumuzun icra edilməsini təmin etməliyik. Lakin burada kiçik bir problem var. Göndərdiyimiz A xarakterlərindən hanslarının(4 bayt – 32) EIP-yə yazıldığını öyrənməliyik.



Bunun üçün metasploit-dən istifadə edirik. Metasploit alətləri arasında məhz bu məqsəd üçün yaradılmış kiçik alətlər var (pattern_create və pattern_offset). İlk olaraq pattern_create ilə 3000 baytlıq pattern yaradırıq. 3000 baytlıq bu pattern müxtəlif xarakterlərdən ibarətdir. Əsas məqsəd eip registeri bizim göndərdiyimiz patternlər ilə dəyişdiyi zaman eip-nin dəyərinin (pattern) hansı offsetdən sonra yazıldığını aşkarlamaqdır.

```
msf5 > ruby pattern_create.rb -l 3000
[*] exec: ruby pattern_create.rb -l 3000
```

Daha sonra patternləri socket üzərindən göndəririk.

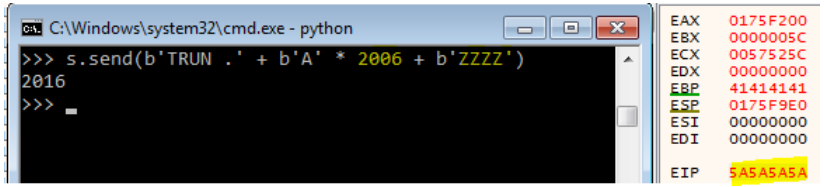


EIP = 0x396F4338. Bundan sonra pattern_offset ilə EIP registerinə yazılan dəyərin hansı offsetdən etibarən yazıldığını öyrənirik.

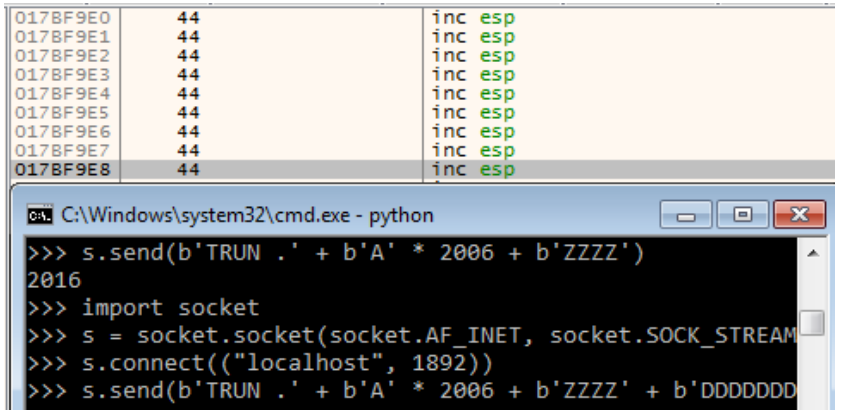
```
msf5 > ruby pattern_offset.rb -q 396F4338
[*] exec: ruby pattern_offset.rb -q 396F4338

[*] Exact match at offset 2006
```

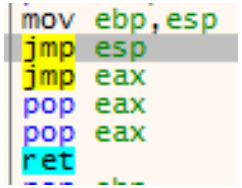
Deməli 2006-cı xarakterdən sonra gələn 4 baytlıq dəyər (32 bit address) EIP registerinə yazılır. Bunu test etmək üçün 2006 baytdan sonra ‘Z’ xarakteri yazaraq göndərirəm.



Bəli gördüyünüz kimi eip registeri 5A5A5A5A (‘ZZZZ’) dəyərlərini aldı. EIP registerini idarə edə bildiyimizə görə bundan sonra qalır faydalı yükümüz. Faydalı yükü qarşı tərəfə ötürmək üçün stack bölgəsindən istifadə edilir. 2006 + 4(eip) məlumatdan sonra göndərdiyimiz məlumat artıq shell kod hissəsidir. Bunu sınaqdan keçirmək üçün 2006 + 4 dən sonra 10 ədəd ‘D’ xarakterini göndərib, stack bölgəsinə baxıram.



Burada da istədiyim nəticəni aldıqdan sonra geriyə qalır pointeri stack bölgəsinə yönləndirmək. Bunun üçün isə proqram yaddaşında JMP ESP instructionu axtarmaq lazımdır. Yaddaşda bu instruction taparaq eip registerini bu instructionun olduğu adres ilə əvəzləyərək CPU-nun stack bölgəsində olan kodları icra etməsini təmin edirik və shell kodumuzicra olunur.



Əlbətdə məsələ bu qədər asan deyil. Xüsusi ilə əməliyyat sistemlərinin və kompilyatorların gətirdiyi bəzi təhlükəsizlik mexanizmləri (DEP, ASLR, SEH, Stack Cookie etc) işləri bir xeyli çətinləşdirir. Lakin zamanla bu mexanizmlərində yayınma metodları araşdırılaraq hazırlanır.

İstinadlar

https://docs.microsoft.com/enus/windows/win32/memory/data-execution-prevention
https://msrc-blog.microsoft.com/2010/12/08/on-theeffectiveness-of-dep-and-aslr/
https://en.wikipedia.org/wiki/Stack_buffer_overflow
https://thegreycorner.com/vulnserver.html