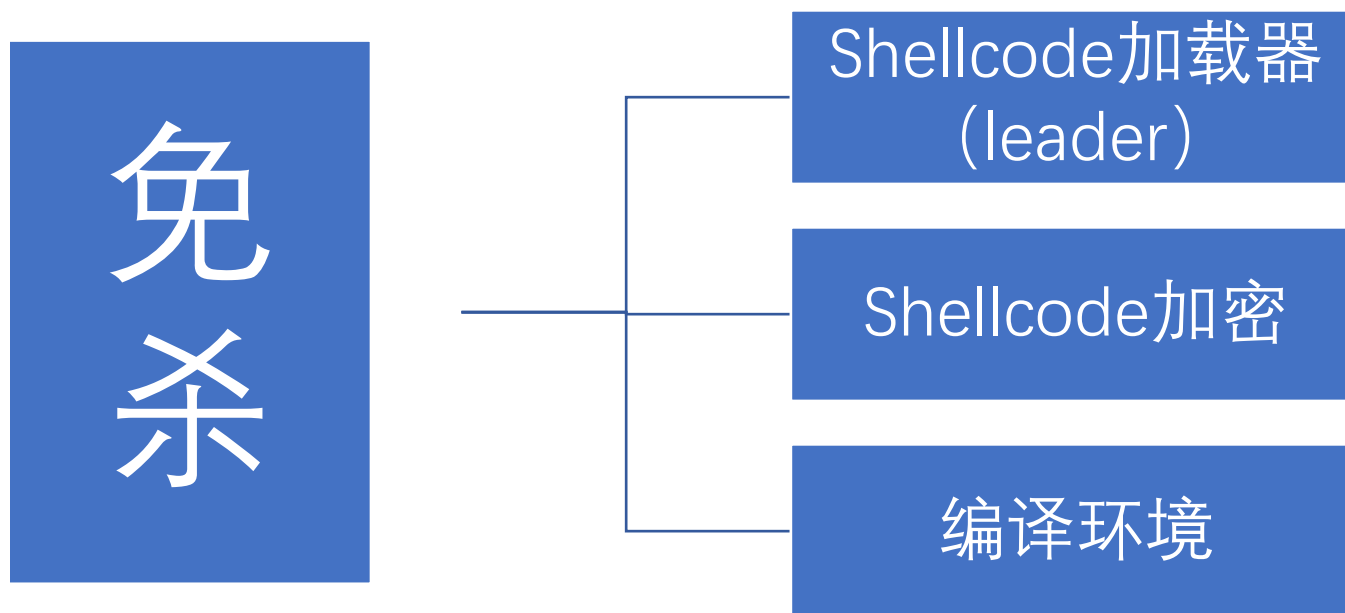


突破主动防御

什么是主动防御

- 主动防御简单来讲，就是一种针对防火墙、注册表、服务、进程、流量等进行监控以及对异常进行拦截的行为。

突破主动防御（免杀）



Shellcode加载器 (loader)

普通加载器：申请动态内存，汇编加载，汇编花指令等。

高级加载器：管道连接（进程间通讯）等。

Shellcode加密

普通加密：base64, xor, AES等。

自定义加密：计算内存偏移，注意32位和64位的指针。

编译环境

- c, c#, c++, python, ruby, go等

突破主动防御的难易程度

简单

- 360
- 腾讯电脑管家

复杂

- 小红伞
- malwarebytes

困难

- Defender (2022年11月15号, 单层自定义加密和管道连接)
- 赛门铁克, 卡巴斯基 (2022年11月15号, 双层自定义加密和管道连接)

进阶免杀（内存）

- 1. 将加密文件加载到字符串存储器时将断点切换到第一个过程后，使用 lea 指令将加密输入数据的有效地址加载到 ESP/RSP 的分配地址中。

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly code with addresses and hex values. The instruction `lea rsp, qword ptr ss:[rsp-50]` is highlighted with a yellow arrow. The comment for this instruction is `[rsp-50]:&"p.\x01"`.
- Registers View:** Shows the state of registers. RAX is 0000000000000000, RBX is 00007FFB497D4758, RCX is 00007FFB49743774, RDX is 0000000000000000, RBP is 00000000013FF380, RSI is 000000000038A000, RDI is 00007FFB497D0740, R8 is 00000000013FF378, R9 is 0000000000000000, R10 is 0000000000000000, R11 is 0000000000000246, R12 is 0000000000000000, R13 is 0000000000000001, R14 is 0000000000000000, R15 is 0000000000000040, and RIP is 00007FFB4977C9E5.
- Call Stack:** Shows the current call stack with the following frames:
 - 1: rcx 00007FFB49743774 ntdll
 - 2: rdx 0000000000000000
 - 3: r8 00000000013FF378 &"p.\x01"
 - 4: r9 0000000000000000
 - 5: [rsp+28] 0000000000000000

2.让加载程序使用有效的解密密钥解密内存流内的加密数据。

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly instructions with addresses and comments. A yellow arrow points to the instruction `mov rax, qword ptr ds:[100038c48]` at address `48:8370 F8 00`, which is labeled "key".
- Registers View:** Shows the state of various registers. The `RAX` register contains the value `0000000015F1900`, which corresponds to the memory address `0000000015F1900` in the comments.
- Comments:** The comments provide context for the assembly instructions, such as `mov rax, qword ptr ds:[100038c48]; &"Empty passphrase"`.

The assembly code includes instructions like `push rbp`, `mov rbp, rsp`, `lea rbp, qword ptr ss:[rbp-20]`, `mov qword ptr ss:[rbp-18], rcx`, `mov qword ptr ss:[rbp-18], r9`, `cmp qword ptr ss:[rbp-18], 1`, `jne komodo.100031848`, `mov rax, qword ptr ss:[rbp-20]`, `mov rdx, qword ptr ss:[rbp-20]`, `mov rcx, rax`, `call qword ptr ds:[rdx+68]`, `mov qword ptr ss:[rbp-20], rax`, `nop dword ptr ds:[rax], eax`, `cmp qword ptr ss:[rbp-20], 0`, `je komodo.100031933`, `nop`, `mov qword ptr ss:[rbp-68], 0`, `cmp qword ptr ss:[rbp-68], 0`, `jne komodo.100031890`, `mov r8, qword ptr ds:[100038c48]`, `mov edx, 1`, `lea rcx, qword ptr ds:[100046]`, `call komodo.10002c4f0`, `mov rcx, rax`, `lea rdx, qword ptr ds:[100031]`, `mov r8, rbp`, `call komodo.100011900`, `nop`, `mov rax, qword ptr ss:[rbp-8]`, `test rax, rax`, `je komodo.100031890`, `mov rax, qword ptr ds:[rax+8]`, `mov dword ptr ss:[rbp-24], eax`, `cmp eax, 35`, `jle komodo.10003184c`, and `mov dword ptr ss:[rbp-24], 38`.

3.由于该输出格式是base64，AV检测是不会查杀的。

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly instructions with their addresses and hex values. The instruction at address 48:112FFFFF is highlighted, showing a call to <JMP.&SetLastError>.
- Registers View:** Shows the state of various registers (RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15) in base64 encoding. The RIP register is also shown, pointing to the instruction at address 48:112FFFFF.
- Registers List:** A list of registers and their values, including RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15, and RIP. The RIP register is highlighted, showing its value as 000000010000E12A.
- Registers Values:** A table showing the values of the registers in base64 encoding. The RIP register is highlighted, showing its value as 000000010000E12A.

A yellow arrow points to the instruction `lea rax, qword ptr ds:[rbx+rdi*1:0x01]` in the assembly view.

4.将恶意有效负载复制到另一个分配的内存地址中。

0000000100001B01	48:89E5	mov rbp, rsp		
0000000100001B04	48:8D6424 90	lea rsp, qword ptr ss:[rsp-70]		
0000000100001B09	48:894D F8	mov qword ptr ss:[rbp-8], rcx	[rbp-8]: "bGF3bGF3"	
0000000100001B0D	48:8955 F0	mov qword ptr ss:[rbp-10], rdx		
0000000100001B11	44:8845 E8	mov byte ptr ss:[rbp-18], r8b		
0000000100001B15	48:C745 C0 00000000	mov qword ptr ss:[rbp-40], 0	[rbp-40]: ""ŷ?\x01"	
0000000100001B1D	90	nop		
0000000100001B1E	48:C745 B0 14000000	mov qword ptr ss:[rbp-50], 14		
0000000100001B26	48:8D15 3B9A0300	lea rdx, qword ptr ds:[10003B568]		
0000000100001B2D	4C:8D4D B0	lea r9, qword ptr ss:[rbp-50]		
0000000100001B31	48:8D4D C0	lea rcx, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B35	41:88 01000000	mov r8d, 1		
0000000100001B3B	E8 207A0000	call komodo.100009560		
0000000100001B40	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B44	C600 90	mov byte ptr ds:[rax], 90		
0000000100001B47	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B4B	C640 01 90	mov byte ptr ds:[rax+1], 90		
0000000100001B4F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B53	C640 02 90	mov byte ptr ds:[rax+2], 90		
0000000100001B57	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B5B	C640 03 90	mov byte ptr ds:[rax+3], 90		
0000000100001B5F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B63	C640 04 90	mov byte ptr ds:[rax+4], 90		
0000000100001B67	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B6B	C640 05 90	mov byte ptr ds:[rax+5], 90		
0000000100001B6F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B73	C640 06 90	mov byte ptr ds:[rax+6], 90		
0000000100001B77	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B7B	C640 07 90	mov byte ptr ds:[rax+7], 90		
0000000100001B7F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B83	C640 08 90	mov byte ptr ds:[rax+8], 90		
0000000100001B87	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B8B	C640 09 90	mov byte ptr ds:[rax+9], 90		
0000000100001B8F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B93	C640 0A 90	mov byte ptr ds:[rax+A], 90		
0000000100001B97	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	
0000000100001B9B	C640 0B 90	mov byte ptr ds:[rax+B], 90		
0000000100001B9F	48:8B45 C0	mov rax, qword ptr ss:[rbp-40]	[rbp-40]: ""ŷ?\x01"	

RIP

RAX 0000000015022

RBX 0000000000000000

RCX 00000000015021

RDY 00000000015022

RBP 00000000013FFC

RSP 00000000013FFE

RSI 00000000013FFE

RDI 0000000000000000

R8 0000000000000000

R9 0101010101010101

R10 00000000334647

R11 0000000081C1AC

R12 0000000000000000

R13 0000000000000000

R14 0000000000000000

R15 0000000000000000

RIP 0000000100001B

RFLAGS 000000000000

ZF 1 PF 1 AF 0

OF 0 SF 0 DF 0

CF 0 TF 0 IF 1

LastError 00000000

LastStatus C0000034

GS 002B FS 0053

ES 002B DS 002B

Default (x64 fastcall)

1: rcx 0000000001502

监管绕过： windows虚拟化与驱动

- windows虚拟化： 系统间通讯、流量抓包、锁屏掉线。
- 驱动： windowsPG保护。

针对人工分析处理问题

- 在我们开发过程中，难免遇见免杀程序被人工分析，一套代码使用一辈子不太现实，毕竟如果人工分析都永远分析不出来，那所有搞查杀的也没必要存在，我们总得给别人一条活路。哪怕曾经的经典熊猫烧香，放在现在，也已经不值一提。我们唯一能做的就是怎么让自己的程序更难被分析。给大家举一个列子吧！这是在国外论坛看到的思路，也使用了近两年，python大家都知道，但是python编译好程序之后，我们可以逆向修改里面的变量以及部分内容，使它不符合python语法的同时，却可以在windows下正常运行，当有程序进行逆向时，程序会直接崩溃，导致程序不可逆。网络安全本就是不断学习的过程，我们与杀软相辅相成，共同进步。学习，借鉴，修改，最后走出自己的路，才是长久之道。

总结

- 免杀无高低，适合的才是最好的，在免杀这条道路上，我们与杀软相辅相成，只有走出自己的路，自己的特点，才可以走得更远。