Авторан на халяву

Введение

Данная статья - результат небольшого исследования **smss** и **csrss** ОС Windows XP SP2. В ходе него было обнаружено несколько возможностей автозагрузки. Все эти техники, пожалуй, уже встречались в том или ином виде в "дикой природе", или, как говорится, itw. Все изложенные методы интересны тем, что запуск происходит на ранних этапах загрузки системы, а удаление исполняемых файлов без соответствующей правки реестра порой приводит к невозможности повторной загрузки ОС (поэтому рекомендую все действия выполнять на виртуальной машине).
Начнем снизу, то есть с **smss**.

Введение в SMSS

SMSS (Session Manager Subsystem Service) - компонент Windows, который отвечает за управление сессиями, как видно из названия, а также производит жизненно необходимые действия на начальных этапах. Довольно поверхностно работа **smss**, как и **csrss**, рассмотрена в известной всем книге «*Внутреннее устройство Microsoft Windows 2000*».
С запуском smss.exe (в **ntos!Phase1Initialization**) связанно 3 BSOD'а: SESSION3_INITIALIZATION_FAILED, SESSION4_INITIALIZATION_FAILED, SESSION5_INITIALIZATION_FAILED. Это наталкивает на мысль, что это довольно ценный компонент системы. Рассмотрим его работу более детально.
Практически сразу после запуска управление получает **smss!SmpInit**, где создается порт **\SmApiPort** и пара потоков **smss!SmpApiLoop**, которые его обслуживают.
Тут, кстати, авторы умной книги немного преувеличивают, утверждая *«… и два потока, ожидающие клиентские запросы (например, на загрузку новой подсистемы или на создание сеанса)»*. На самом деле, никаких новых подсистем **smss** не загружает, есть лишь сообщение **SmLoadDeferedSubsystemApi**, получение которого приводит к подгрузке подсистем, которые содержатся в списке **smss!SmpSubSystemsToDefer**. Данный список, как и другие не менее интересные списки, заполняется в результате вызова обработчика, указанного в таблице **smss!SmpRegistryConfigurationTable**, которая передается в **RtlQueryRegistryValues**.
Рассмотрим ключи, имена которых встречаются в данной таблице.

1) **ProtectionMode**. Влияет на установку прав доступа к **\SmApiPort**'у;
2) **AllowProtectedRenames**. Если установлен, то разрешает отложенное перемещение файлов, защищенных WFP;
3) **ObjectDirectories**. Приводит к созданию каталогов (**NtCreateDirectoryObject**) с указанными именами;
4) **BootExecute**. Указание приложения в этом ключе приводит к его запуску. Autoruns выводит список приложений, указанных в этом ключе. Широко известный и используемый ключ, используемый для запуска Native приложений. При указании имени Native-приложения, которое необходимо загрузить, кроме идентификатора **autocheck** (используется при указании autochk в этом списке) возможны идентификаторы **async** (приводит к тому, что система не ожидает завершения запускаемого процесса) и **debug** (приводит к установке **ProcessParameters->DebugFlags = TRUE**).
   http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx - кадр из Microsoft немного пишет по теме;
5) **SetupExecute**. Аналогичный BootExecute, только приложения вызываются после вызова **NtInitializeRegistry**. А также при получении сообщения **SmStartCsr**;
6) **PendingFileRenameOperations, PendingFileRenameOperations2**. Содержат информацию о файлах, которые должны быть переименованы после перезагрузки системы;
7) **PagingFiles, DOS Devices, ExcludeFromKnownDlls, KnownDlls, Environment**. Неинтересные в контексте данной статьи ключи, назначение которых интуитивно понятно;
8) **Subsystems, Require, Optional, KMode**. Отвечают за запуск подсистем и будут рассмотрены более детально ниже;
9) **Execute**. Позволяет запустить приложения. Более подробно будет рассмотрен ниже.

Данный список может варьироваться от системы к системе, хоть различия и не значительны.
Теперь рассмотрим способы автозагрузки исходя из полученной информации.

Загрузка через BootExecute SetupExecute.
Создадим тестовое Native приложение, которое будет выводить строку на экран приветствия с помощью **NtDisplayString**.

```
#pragma comment(linker,"/ENTRY:NtProcessStartup")

#define _X86_
#include <ntddk.h>
#include <windef.h>

extern "C"{
	NTSYSAPI NTSTATUS NTAPI    NtDisplayString(IN PUNICODE_STRING String );
	NTSYSAPI NTSTATUS NTAPI NtTerminateProcess(IN HANDLE ProcessHandle OPTIONAL,IN
NTSTATUS ExitStatus);
};
void APIENTRY NtProcessStartup()
{
	UNICODE_STRING usMess;
	RtlInitUnicodeString(&usMess,L"Hello, im pretty fine dummy native application\n");
	NtDisplayString(&usMess);
	NtTerminateProcess(NtCurrentProcess(),0);
}
```

Полученный файл скопируем в директорию *system32*. В реестре в разделе Session Manager редактируем/создаем параметр *BootExecute/SetupExecute* типа *REG_MULTI_SZ*, содержащий строку **native**, где **native** – имя программы. После чего производим перезагрузку, наблюдая при загрузке картину типа этой:
(1.jpg)

Загрузка через Execute.
Принцип загрузки с помощью этого ключа похож на принцип, используемый при загрузке с помощью BootExecute. Но есть и нюанс. Прежде чем запустить все приложения, которые присутствуют в списке, исполняется следующий код:

```
 v10 = &SmpExecuteList;
 if ( SmpExecuteList == (_DWORD)&SmpExecuteList )//проверка пуст ли список
 {
  RtlInitUnicodeString(usIC, L"winlogon.exe");
  InitialCommandBuffer = 0;
  LdrQueryImageFileExecutionOptions((int)usIC, (int)L"Debugger", 1, (int)&InitialCommandBuffer, 512, 0);
  if ( InitialCommandBuffer )
  {
   _wcscat(&InitialCommandBuffer, L" ");
   _wcscat(&InitialCommandBuffer, usIC->Buffer);
   RtlInitUnicodeString(usIC, &InitialCommandBuffer);
   v10 = &SmpExecuteList;
  }
 }
 else
 {
  v14 = smpexecOfs;
  v15 = *(_DWORD *)&smpexecOfs->usStr.Length;
  DestinationString.HighPart = -1;
  *(_DWORD *)&usIC->Length = v15;
  usIC->Buffer = v14->usStr.Buffer;
  DestinationString.LowPart = -50000000;
  NtDelayExecution(0, &DestinationString);
 }
```

Тобишь присутствует некая структура UNICODE_STRING, указатель на которую передаётся из **smss!_main** в **smss!SmpInit -> smss!SmpLoadDataFromRegistry -> smss! SmpLoadSubSystemsForMuSession**. Эта структура заполняется на данном участке кода и команда, которая записана туда после возврата из **smss!SmpInit** будете передана в **smss!**

SmpExecuteInitialCommand на исполнение.
В случае отсутствия **Execute** параметра эта строка будет содержать «*winlogon.exe*» либо, если присутствует параметр **Debugger** в ключе реестра *HKLM\Software\Microsoft\Windows NT\Current Version\Image File Execution Options\winlogon.exe*, то строка «*winlogon.exe»* будет присоединена к значению этого параметра.
В случае присутствия **Execute** строка будет инициализирована последним элементом, указанным в данном параметре. После чего будет выполнена 5 секундная задержка, связанная с выполнением **csr**. Также не будет давать желаемого результата вызов **NtDisplayString**, т.к. на момент прохода списка подсистемы уже запущены. Поэтому немного переделаем тестовый пример:

```
void APIENTRY NtProcessStartup()
{
	while(1)
		DbgPrint("Hello\n");
}
```

А значение параметра **Execute** установим:
async native.exe
winlogon.exe
Идентификатор **async** необходим для того, чтоб система не ожидала завершения приложения, которое не рассчитано на то, чтобы завершиться.

Автозагрузка добавлением в список подсистем.
Перейдем к сабкею **SubSystems.**
Здесь все предельно ясно. Есть два параметра – **Required** и **Optional**. В них содержаться имена подсистем, которые необходимо запустить. Разница между **Required** и **Optional** в том, что подсистемы, попадающие во второй список, запускаются по запросу, а не сразу. Поэтому будем использовать параметр **Required** для добавления своей «подсистемы».
После добавления имени подсистемы необходимо создать параметр с именем, которое совпадает с именем «подсистемы» и значением, которое содержит имя сервера подсистемы.
Но, даже проделав данные операции с реестром, максимум, чего мы добьемся – стабильное зависание на этапе загрузки подсистем. Это связано с тем, что в отличие от старта из вышеописанных ключей, для запуска подсистем используется не **smss!SmpExecuteImage**, а **smss!SmpLoadSubSystem**, которая также вызывает **smss!SmpExecuteImage** но кроме того добавляет в список подсистем информацию о нашей подсистеме. Также запись о подсистеме содержит описатель события, на котором ожидает **smss** после запуска подсистемы. Событие переводится в сигнальное состояния после того, как подсистема произведет свою инициализацию и сообщит об этом коннектом на порт **\SmApiPort**.
Новый исходный код тестового приложения:

```
#pragma comment(linker,"/ENTRY:NtProcessStartup")
#define _X86_
#include <ntddk.h>
#include <windef.h>

extern "C"{
	NTSYSAPI NTSTATUS NTAPI    NtDisplayString(IN PUNICODE_STRING String );
	NTSYSAPI NTSTATUS NTAPI NtTerminateProcess(IN HANDLE ProcessHandle OPTIONAL,IN
NTSTATUS ExitStatus);
	NTSYSAPI NTSTATUS NTAPI    NtConnectPort(OUT PHANDLE ClientPortHandle,
		IN PUNICODE_STRING ServerPortName, IN PSECURITY_QUALITY_OF_SERVICE
SecurityQos,
		IN OUT /*PLPC_SECTION_OWNER_MEMORY*/ PVOID ClientSharedMemory OPTIONAL,
		OUT /*PLPC_SECTION_MEMORY*/ PVOID ServerSharedMemory OPTIONAL,
		OUT PULONG MaximumMessageLength OPTIONAL,
		IN OUT PVOID ConnectionInfo OPTIONAL,
		IN OUT PULONG ConnectionInfoLength OPTIONAL );

};

typedef struct _SBCONNECTINFO {
	ULONG SubsystemImageType;
	WCHAR EmulationSubSystemPortName[120];
} SBCONNECTINFO, *PSBCONNECTINFO;
```

```
#define OUR_SUBSYSTEM_TYPE 0x0DEAD

NTSTATUS ConnectToSmss(OUT PHANDLE SmApiPort )
{
	NTSTATUS st;
	UNICODE_STRING PortName;
	ULONG ConnectInfoLength;
	SBCONNECTINFO ConnectInfo;
	SECURITY_QUALITY_OF_SERVICE DynamicQos =
	{
		sizeof(DynamicQos), SecurityImpersonation, SECURITY_DYNAMIC_TRACKING, TRUE
	};
	RtlInitUnicodeString(&PortName,L"\\SmApiPort");
	ConnectInfoLength = sizeof(SBCONNECTINFO);
	ConnectInfo.SubsystemImageType = OUR_SUBSYSTEM_TYPE;
	st =
NtConnectPort(SmApiPort,&PortName,&DynamicQos,NULL,NULL,NULL,(PVOID)&ConnectInfo,&Connect
InfoLength);
	return st;

}

void APIENTRY NtProcessStartup()
{
	HANDLE SmApiPort;
	ConnectToSmss(&SmApiPort);

	while(1)
		DbgPrint("Hello\n");

	NtTerminateProcess(NtCurrentProcess(),0);
}
```

Необходимо добавить параметр **native** в **SubSystems**, значением которого будет путь к **native.exe**. А также в параметр **Required** добавить строку **native**, после чего выполнить перезагрузку.
Собственно это все способы автозагрузки, которые бросились мне в глаза во время исследования **smss.exe.** Также я заметил то, что одна из подсистем, которые загружаются, csrss, имеет довольно интересный набор параметров, что натолкнуло меня на мысль, что там также есть возможность авторана.

CSRSS и автозапуск
На самом деле csrss.exe – абсолютно неинтересная штуковина, самое полезное действие которой – вызов **csrsrv!CsrServerInitialization**. Если в **smss** все настройки содержались в реестре, то **csrss** берет настройки из командной строки, которая тоже указана в реестре и выглядит примерно так:
%SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,3072,512 Windows=On SubSystemType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=winsrv:ConServerDllInitialization,2 ProfileControl=Off MaxRequestThreads=16
Парсинг этой строки производится в **csrsrv!CsrParseServerCommandLine.** Параметры, которые начинаются с “ServerDLL=”, содержат в себе информацию о подгружаемых библиотеках в виде x:y,z где x – имя библиотеки, y – имя процедуры инициализации, z – номер этой библиотеки.
Загрузка этих библиотек происходит в **csrsrv!CsrLoadServerDll** следующим образом:

1) Проверяется номер библиотеки
   mov esi, [ebp+ModuleNum]
   cmp esi, 4
   jb short l_ok
2) Проверяется не была ли библиотека с этим номером уже запущена
   l_ok:
   xor edi, edi
   cmp _CsrLoadedServerDll[esi*4], edi
   jz short l_ok2
3) Загружается библиотека средствами **ntdll!LdrLoadDll**
4) Инициализируется структура **CSR_SERVER_DLL**
5) Вызывается процедура инициализации, указанная для данной библиотеки. В качестве

параметра функция получает указатель на инициализированную структуру **CSR_SERVER_DLL** для внесения дополнительной информации
push esi
call [ebp+var_20]

6) Указатель на эту структуру сохраняется в массив **CsrLoadedServerDll**
mov eax, [esi+14h] <- здесь содержится номер библиотеки после инициализации
mov _CsrLoadedServerDll[eax*4], esi

Ошибка на любом из этапов – синий экран.
В случае, если мы хотим использовать командную строку как возможность автозапуска очевидно, что необходимо присвоить своей библиотеке номер < 4. *Trojan.Win32.SubSys,* дабы решить эту проблему, подменял запись одного из winsrv на свою. Я же предлагаю добавить свою запись. Часть строки инициализации, отвечающая за подгрузку библиотек, будет выглядеть следующим образом:
ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=native:MyServerDllInitialization,2 ServerDll=winsrv:ConServerDllInitialization,2
Но тут возникает проблема с тем, что после библиотеки с номером 2 следует ещё одна библиотека с номером 2. То есть, для успешной загрузки второй библиотеки первая должна загрузиться так, чтобы в _CsrLoadedServerDll[2*4] остался 0. Возможность проделать это можно обнаружить внимательно посмотрев на процесс загрузки ещё раз. Номер библиотеки попадает в структуру **CSR_SERVER_DLL** во время её инициализации, но после этого указатель на структуру передается в процедуру инициализации, где можно подменить номер библиотеки, после чего на 6ом шаге указатель на структуру попадет по адресу **CsrLoadedServerDll+4*новый_номер_библиотеки.**
Исходный код **native.dll,** которая реализует подобный механизм загрузки:

```
#pragma comment(linker,"/ENTRY:DllEntry")
#pragma comment(linker, "/SECTION:.text,EWR")
#pragma comment(linker, "/SECTION:.rdata,EWR")
#define _X86_
#include <ntddk.h>
#include <windef.h>

//
//Структура, описывающая библиотеку
//
typedef struct _CSR_SERVER_DLL {
	ULONG Length;
	HANDLE CsrInitializationEvent;
	STRING ModuleName;
	HANDLE ModuleHandle;
	ULONG ServerDllIndex;// номер библиотеки
	ULONG ServerDllConnectInfoLength;
	ULONG ApiNumberBase;
	ULONG MaxApiNumber;
	union {
		ULONG ApiDispatchTable;
		ULONG QuickApiDispatchTable;
	};
	PBOOLEAN ApiServerValidTable;
	PSZ *ApiNameTable;
	ULONG PerProcessDataLength;
	ULONG PerThreadDataLength;
	ULONG ConnectRoutine;
	ULONG DisconnectRoutine;
	ULONG AddThreadRoutine;
	ULONG DeleteThreadRoutine;
	ULONG InitThreadRoutine;
	ULONG ExceptionRoutine;
	ULONG HardErrorRoutine;
	PVOID SharedStaticServerData;
	ULONG AddProcessRoutine;
	ULONG ShutdownProcessRoutine;
	ULONG ApiDispatchRoutine;
} CSR_SERVER_DLL, *PCSR_SERVER_DLL;
typedef BOOL (*PFNNOTIFYPROCESSCREATE)(DWORD,DWORD,DWORD,DWORD);
```

```
typedef BOOL (*PUSER_THREAD_START_ROUTINE)(DWORD);

// небольшая полезная нагрузка
extern "C" {
	void BaseSetProcessCreateNotify(IN PFNNOTIFYPROCESSCREATE ProcessCreateRoutine);
};
BOOL ProcessCreateRoutine(HANDLE UniqueProcess,HANDLE UniqueThread, DWORD shit, DWORD
dwFlags);
PCSR_SERVER_DLL LoadedServerD;

//
//Процедура инициализации, вызываемая после
//заполнения вышеупомянутой структуры. Должна присутствовать в экспорте dll
//
NTSTATUS MyServerDllInitialization(PCSR_SERVER_DLL LoadedServerDll)
{
	//
	//Сигнатурный поиск CsrLoadedServerDll
	//
	LoadedServerD=LoadedServerDll;
	ULONG* startaddr=(ULONG*)&LoadedServerDll;
	startaddr--;
	unsigned char*search=(unsigned char*)*startaddr;
	//
	//Ищем
	// mov     _CsrLoadedServerDll[eax*4], esi
	//
	while ((*search!=0x89)||(*(search+1)!=0x34)) search++;
	ULONG CsrLoadedServerDll=*(ULONG*)(search+3);

	//
	//подсчитываем новый номер так, чтобы
	//mov     _CsrLoadedServerDll[eax*4], esi
	//записало LoadedServerDll не в _CsrLoadedServerDll[2*4]
	//а в поле LoadedServerDll->CsrInitializationEvent,
	//которое выбрано случайно.
	//
	LoadedServerDll->ServerDllIndex=(ULONG)&LoadedServerDll->CsrInitializationEvent/4-
CsrLoadedServerDll/4;

	//
	//На всякий случай
	//
	LoadedServerDll->PerProcessDataLength = 0;
	LoadedServerDll->PerThreadDataLength = 0;

	//
	//Полезная нагрузка
	//
	BaseSetProcessCreateNotify((PFNNOTIFYPROCESSCREATE)&ProcessCreateRoutine);
	return 0;
}


int APIENTRY DllEntry(HINSTANCE h_module, DWORD dw_reason, LPVOID p_reserved)
{
	DbgPrint("We're in\n\n\n");
	return 1;
}

BOOL ProcessCreateRoutine(HANDLE UniqueProcess,HANDLE UniqueThread, DWORD shit, DWORD
dwFlags)
{
	DbgPrint("New Process created PID=%08X TID=%08X\n",UniqueProcess,UniqueThread);
	return true;
}
```

Пара слов о полезной нагрузке.
basesrv.dll, которая также подгружается в пространство процесса **csrss,** экспортирует функцию **BaseSetProcessCreateNotify,** которая инициализирует глобальную переменную **basesrv! UserNotifyProcessCreate** переданным в данную функцию значением. При создании пользовательского процесса происходит нотификация **basesrv**, которая после завершения инициализации вызывает функцию, указатель на которую находится в **basesrv! UserNotifyProcessCreate**. Более глубоко в эту сторону копнул **blast**
http://virustech.org/f/viewtopic.php?id=25

На этом заканчиваю свой обзор.
Спасибо коллективу **cih.[ms]** за моральную поддержку во время исследования.