# Malware Tradecraft

Malware evasion and stealth techniques

Start Week 7 →


```
f4a1 8bd0 23cf 79e0 1174 c39a 26d7 b901
9f02 651a 88d3 1ab5 ef4c a918 6b7c 3ffe
3a1e e823 4ff9 02d8 becb 5f90 a4d2 67aa
f4a1 8bd0 23cf 79e0 1174 c39a 26d7 b901
```

# Quiz

Good luck.

31

MINUTES

− +

*Ends at 5:20 PM*

# Malware Tradecraft

Unlike most software, malware authors apply techniques to evade detection, delay analysis, hide software functionality, bypass security measures, and persist on systems without the user's knowledge.

- Hide behavior from the user
- Hide behavior from anti-virus
- Weaken protections like firewall rules
- Prevent uninstallation
- Slow down the reverse engineering process

# Other Uses

These techniques are mostly used by malware, but can sometimes be seen in other software.

- License checking to authorize software usage
- Anti-cheat systems for games
- Protecting proprietary protocols (e.g. Skype)
- Protect proprietary security features from malware

# Example 1: Skype

Skype went to great lengths to frustrate potential developers of open-source compatible clients.

- Custom packer, erases and re-writes its own import table at runtime
- Portions of the binary are stored ciphered and decrypted before execution
- Checksum’s its own code to detect modifications like breakpoints
- Custom network-layer RC4-based obfuscator
- Applies custom arithmetic compression to packets

https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf

# Example 2: Patch Guard

Secretive Windows security feature meant to protect the integrity of the kernel. If it detects kernel modifications that violate policy it may crash the system with a `CRITICAL_STRUCTURE_CORRUPTION` exception.

- Periodically hashes things like kernel text, page tables to detect issues
- Code and data are encrypted at runtime
- Symbol names are misleading
- In-memory layout randomizes across boots and builds to frustrate modification
- Runs on multiple threads that trigger on irregular intervals
- Confusing control flow (e.g. using exceptions intentionally)
- May crash the system on upon detecting breakpoints or timing anomalies

# Objectives

Today we'll cover three broad categories of techniques for frustrating analysis.

- Anti-static analysis (import hiding, string obfuscation, packing, code obfuscation)
- Anti-dynamic analysis (timing, anti-debug tricks, anti-VM tricks)
- Hiding (dll injection, process hollowing, function hooking)

# Hiding Imports

Remember last class when we looked at the import table for cross references? Malware can hide imports by loading the libraries and finding the target symbols dynamically.

```c
#include <windows.h>

typedef BOOL (WINAPI *LP_ISWOW64PROCESS)(HANDLE, PBOOL);

int main(void) {
  HMODULE kernel32 = LoadLibraryA("kernel32.dll");
  LP_ISWOW64PROCESS isWow64 = (LP_ISWOW64PROCESS)
      GetProcAddress(kernel32, "IsWow64Process");

  BOOL wow64 = FALSE;
  if (isWow64 && isWow64(GetCurrentProcess(), &wow64) && wow64) {
    // Branch when running under emulation (common sandbox target)
    ExitProcess(0);
  }
  return 0;
}
```

In this example we won't see the import. What might we see during triage, instead?

# Hiding Import Strings

API hashing removes plaintext strings while keeping dynamic resolution flexible.

- Malware iterates DLL export tables, hashing each name, and compares against a hard-coded value
- Hash algorithms are short (XOR, rotate, djb2-style) to minimize footprint

```c
DWORD hash_name(const char *name) {
  DWORD hash = 0x811C9DC5;  // FNV-1a seed
  while (*name) {
    hash ^= (BYTE)(*name++ | 0x20); // case-insensitive
    hash *= 0x01000193;
  }
  return hash;
}

FARPROC resolve_by_hash(HMODULE module, DWORD target_hash) {
  PIMAGE_EXPORT_DIRECTORY exports = get_export_directory(module);
  DWORD *names = RVA_TO_PTR(module, exports->AddressOfNames);
  WORD *ordinals = RVA_TO_PTR(module, exports->AddressOfNameOrdinals);
  DWORD *functions = RVA_TO_PTR(module, exports->AddressOfFunctions);

  for (DWORD i = 0; i < exports->NumberOfNames; ++i) {
```

# String Munging

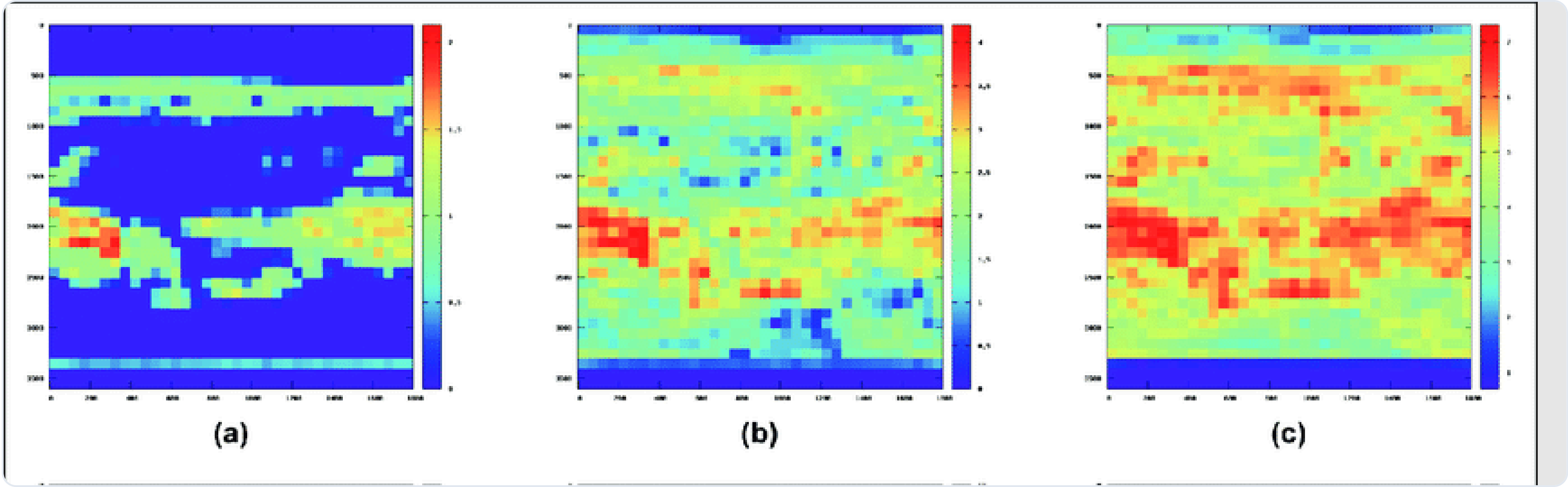
We may want to hide other strings from analysis as well. We can do this by encrypting strings and decrypting them at runtime.

```c
static const BYTE blob[] = {0x1F, 0x0C, 0x47, 0x09, 0x01, 0x5A, 0x00};

void decode(char *dst) {
  for (size_t i = 0; blob[i] != 0; ++i) {
    BYTE key = (BYTE)(0x55 + (i * 7));
    dst[i] = (char)(blob[i] ^ key);
  }
}

int main(void) {
  char buffer[8] = {0};
  decode(buffer);
  printf("%s\\n", buffer);   // Prints "cmd.exe"
  SecureZeroMemory(buffer, sizeof(buffer));
  return 0;
}
```

# Packing

To hide all code/data, a malware author may use a packer. These will encrypt the entire program except for a stub at the entry point, which decrypts the other sections before execution.

- Few functions found in a large binary
- Large data region that appears encrypted
- Large high-entropy region



# Source Obfuscation

Source obfuscation is use for malware in interpreted langauges.

- Minifiers that strip whitespace/comments and renamne identifiers
- AST transformers reorder logic, split strings, etc.
- Example: not all minification is malicious
- Example: NPM Supply Chain Attack

# Code Transformations

For compiled langauges source transformations can still be useful to obfuscate program logic.

- Example: Obfuscated C Competition
- Example: movfuscator
- In practice malware authors will write LLVM transformations to obfuscate compiled code
  - Allows for easy polymorphism: malware looks different every compilation which makes it hard to signature
  - Example: Chris Domas

# Anti-Debugging

Malware might probe its environment to check if a debugger is present.

- **Timing**: compare `QueryPerformanceCounter`, `GetTickCount`, `RDTSC` values before/after loops
- **API checks**: `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, `NtQueryInformationProcess`, `ptrace` on linux
- **Exception abuse**: `INT 2D`, `OutputDebugString`, single-step manipulation
- **Hardware**: check `CPUID` hypervisor bit, look for `vmware` or `vbox` devices
- **Privilege**: attempt `SeDebugPrivilege` escalation, bail if unavailable

# Beyond Debuggers

How might malware detect it's in a virtual machine?

- **Sandbox detection**: look for short uptime, few processes, synthetic usernames, RAM size
- **I/O tricks**: write to temp files or registry keys and verify persistence (VM snapshots revert state)
- **Network timing**: enforce C2 beacons that detect latency spikes caused by instrumentation

# Lab 1

https://hacs408e.umd.edu/labs/week-07/lab-1/

86

MINUTES

− +

*Ends at 6:15 PM*

# DLL Injection & Side-Loading

Malware often tries to hide inside legitimate programs to borrow trust and stay persistant. One way to do this is to force legitimate programs to load a malicious dynamic library.

- Drop a malicious DLL next to a trusted binary that auto-loads plug-ins or helpers
- Abuse DLL search order so a spoofed name (e.g., `version.dll`) is picked before the real dependency
- Reflectively load the DLL from memory to avoid touching disk after initial compromise
- Legitimate UI/process masks malicious behavior while inheriting its privileges

# Remote Shellcode Injection

Injecting shellcode into another program.

```c
#include <windows.h>

int inject_shellcode(DWORD pid, const unsigned char *shellcode, SIZE_T size) {
  HANDLE process = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
                               PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_VM_READ,
                               FALSE, pid);
  if (!process) {
    return 1;
  }

  LPVOID remote = VirtualAllocEx(process, NULL, size, MEM_COMMIT | MEM_RESERVE,
                                 PAGE_EXECUTE_READWRITE);
  if (!remote) {
    CloseHandle(process);
    return 2;
  }

  if (!WriteProcessMemory(process, remote, shellcode, size, NULL)) {
    VirtualFreeEx(process, remote, 0, MEM_RELEASE);
    CloseHandle(process);
```

# Other Techniques

Novel techniques are common so you'll have to investigate API usage. What does this do?

```c
#include <windows.h>

int main(void) {
  const char dllPath[] = "C:\\\\Windows\\\\Temp\\\\payload.dll";
  DWORD targetPid = 4321; // replace with target process ID

  HANDLE process = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
                               PROCESS_VM_OPERATION | PROCESS_VM_WRITE,
                               FALSE, targetPid);
  if (!process) {
    return 1;
  }

  LPVOID remote = VirtualAllocEx(process, NULL, sizeof(dllPath),
                                 MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
  if (!remote) {
    CloseHandle(process);
    return 2;
  }
```

# Hooking Fundamentals

Another common malware technique is function hooking, where malware modifies a process to call back to the malware when certain functionality is hit.

With what we've learned in class, what might be some approaches to this?

- Replace portions of the binary with malicious shellcode
- Inserting a `jmp` instruction at the beginning of a function to redirect execution
- Modifying the PLT/GOT to point to malicious functions

# Lab 2

https://hacs408e.umd.edu/labs/week-07/lab-2/

171

MINUTES

−  +

*Ends at 7:40 PM*

# Midterm

- Midterm presentations are next week!