How to develop an unpacker

The StarForce case

Date 07/04/2017
At Sthack security conference in Bordeaux
By Eloi Vanderbeken
Whoami

- Eloi Vanderbeken IRL
- @elvanderb on twitter

Working for Synacktiv:

- Offensive security company (pentest, red team, vuln hunting, exploitation etc.)
- If there is software in it, we can own it :)
- We are recruiting!
Packers

- Several types of packers
  - Malware packers: often very simple, just used to bypass AV
  - Compressor: also very simple, just used to reduce binaries size (UPX)
  - Protectors: need to resist to skilled reversers / crackers

- Protectors
  - Wrap an existing program into another one
  - Offer APIs to interact with the packer (licensing, protected variables etc.)
  - New program is harder to study (Anti-X, virtualization, etc.)
  - The protection should not be easy to remove → protection and original program must be entangled
Offensive information security?
Yes!

- Some vuln^winteresting programs are protected by protectors
- You won’t be able to reverse or fuzz them without unpacking them
- Unpacking is the sum of numerous useful skills for a vuln hunter
  - reversing, automation, Windows internals, PE format, etc.
- It’s fun, you fight against someone trying to block you
Our target: StarForce

- What we won’t cover: StarForce Disc
  - Infamous protection used in 2000-2007
  - Used a ring0 driver and virtualization
  - Resisted to crackers for 420 days (!!!)

- What we’ll see: StarForce ProActive
  - Lighter protection (no r0, no VM)
  - Includes licensing tools
  - Used to protect a lot of Shareware
  - A lot simpler than the older one but still interesting :)
Our unpacker: Astroboon
Architecture of our unpacker

- DLL injected in the targeted process
  - No debug API
  - No memory translation needed
  - Direct access to several information (PEB, registers)
- Coded in C
  - And some inlined ASM
  - 1200 lines of StarForce specific code
- (Almost) no external dependencies
  - It uses BeatriX LDE but it also includes my own disassembler so I could drop the LDE
  - Includes a PE parser, a PE dumper, an import fixer, a code hooker, a disassembler, etc
Organisation of the slides

- For each protection
  - Description of the protection
  - Description on how it’s implemented by various protectors
  - How to bypass it in the StarForce case
  - How to implement the automatic bypass in our unpacker

- At each step, if you have any question, please ask :)
Part 1: layers

Original Entry Point (OEP)

Time line

PE Before

Packer’s code

PE After
Layers: what we need to do

- Find the OEP
  - Signatures of common RT entry points
  - Hooks on APIs commonly used at the entry point (GetCommandLine)
  - Examination of the call stack and code xrefs
  - etc.

- Dump the process
  - LordPE / ImpRec (a little bit outdated now 😊)
  - Scylla (open source !)
  - BaDu (Baboon’s Dumper (yes, I know))
Layers: How to automatically find the OEP

- Change pages rights
  - Remove the eXecution right

- Make sure they are not restored
  - Hook VirtualProtect

- Catch the exceptions
  - We use Vectored Exception Handlers
  - We could put a hook on KiUserExceptionDispatcher…
  - … but some packers will detect this

- When the process tries to execute one of the first sections: we are at the OEP
Layers: How to automatically find the OEP

```c
VirtualProtectAddr = (PBYTE)GetProcAddress(GetModuleHandleA("Kernel32"), "VirtualProtect");
hookFun(VirtualProtectAddr, (PBYTE)HookedVirtualProtect, (FARPROC*)&OrigVirtualProtect);
vectoredHandler = AddVectoredExceptionHandler(0, ProtectionFaultVectoredHandler);

LONG CALLBACK ProtectionFaultVectoredHandler(PEXCEPTION_POINTERS ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_GUARD_PAGE_VIOLATION)
    {
        DWORD address = ExceptionInfo->ExceptionRecord->ExceptionInformation[1];
        DWORD eip = ExceptionInfo->ContextRecord->Eip;
        DWORD oldProtect;

        OrigVirtualProtect((PBYTE)textaddress, textsize, PAGE_EXECUTE_READWRITE, &oldProtect);

        if ((eip == address) && (address >= (DWORD)textaddress) && (address < (DWORD)(textaddress+textsize)))
        {
            MessageBoxA(0, "OEP LOL", "OEP LOL", 0);
            Dump(...);
        }
    return EXCEPTION_CONTINUE_EXECUTION;
}
return EXCEPTION_CONTINUE_SEARCH;
```
Part 2: API redirection

GetCurrentProcessId

Kernel32.dll

Obfuscated API code

Allocated Memory
API Redir: what we need to do

- **Find the IAT**
  - Find all the call [XXX] / jmp [XXX]
  - Search for API addresses above and between the min and max addresses

- **Fix redirections**
  - Very protector specific, different kind of redirections
  - Some of them includes special protections in them (SecuROM triggers)

- **Two main approaches:**
  - Hook the redirection mechanism
    - We will have the real API addresses...
    - But need to find the redirection mechanism (signatures, heuristics etc.)
  - Try to recover the original API address from the redirection

- **Once the original addresses are recovered, rebuild the IAT**
  - ImpRec / ChimpREC (a little bit outdated)
  - Scylla
  - BINI: BINI Is Not ImpRec (No Baboon in this name!)
API Redir: StarForce case

- **Addresses in the IAT point to obfuscated version of the original API**
  - No direct redirection in the code (call [API addr] replaced by call REDIRECTION for example)
  - No destruction of the IAT (all the addresses are at their original place)

- **Obfuscated version is created on the fly**
  - Even the API with known behavior (GetCurrentProcessId, GetCurrentProcess, GetProcessHeap, etc.)

- **Sometimes the entire API is rewritten**
  - no final jump to the original code to help us

- **~ 20 obfuscation rules**
  - cmc / cmc = nop
  - push X / xchg [esp], Y = push Y
  - etc.
Astroboon approach

- Construct a canonical representation
  - Disassemble the code
  - Stop when we encounter a RET
  - Follow the unconditional JMPs, not the JCC
  - Don’t enter the calls
  - Deobfuscate the produced trace

- If the canonical representation of an obfuscated code matches the one of an API → WIN

- But we can have multiple matches in multiple DLLs
  - We can use adjacent addresses to solve this problem
  - Adjacent addresses → same DLL
Astroboon approach - cont’d

```c
else if ((i+2 < length) &&
         (StrictInsCmp(&instructions[i], &xchg)) &&
         (StrictInsCmp(&instructions[i+1], &push)) &&
         (StrictInsCmp(&instructions[i+2], &xchg)) &&
         (ArgCmp(TYPE_REG, &instructions[i].Arg1, &instructions[i+2].Arg1)) &&
         (ArgCmp(TYPE_REG, &instructions[i].Arg2, &instructions[i+1].Arg1)) &&
         (ArgCmp(TYPE_REG, &instructions[i].Arg2, &instructions[i+2].Arg2)))
{
    retins[j].instruction = INS_PUSH;
    retins[j].ArgT1 = instructions[i].ArgT1;
    retins[j].PG1 = retins[j].PG2 = retins[j].PG3 = retins[j].PG4 = 0;
    CopyMemory(&retins[j].Arg1, &instructions[i].Arg1, sizeof(ARG));
    retins[j].ArgT2 = 0;
    i+=2;
    j++;
}
```
Part 3: Code redirection

Anti-dump + obfuscation
Code redirections: how to fix this

- **Find all the redirections**
  - Find all the call / jmp / jcc instructions which point to the StarForce section

- **Fix the redirections**
  - Depends on the protector
  - Often based on tracing methods
Astroboon approach

- A code is always used a little bit before jumping to the original code
- It doesn’t change between versions
  - Easy to put a sig on it
- All we have to do is set a HBP on it, jump on the redirection and let StarForce do the redirection for us.
  - Modify debug registers with SetThreadContext
  - Make sure our HBP cannot be detected with a SEH by clearing the DRs in our VEH and restoring them via a hook on ZwContinue
- To find the final jump we trace the code step by step by setting the Trap Flag
  - To make sure it’s not detected/cleared with a PUSHFD/POPFD, we clear/set the Trap Flag in the stack when we detect those instructions after/before their execution.
Part 4: StarForce MISC

- StarForce tries to detect VMs
  - Under VirtualBox, just clear the registry key HKLM\HARDWARE\DESCRIPTION\System\VideoBiosVersion
- StarForce has a watchdog thread that detects debuggers and patches
  - Just kill it before starting to reconstruct the executable
- StarForce uses the (non-reversible) ThreadHideFromDebugger thread information class to... hide threads from the debugger
  - Hook NtSetInformationThread and block the calls
Part 5: MISC MISC

- When your reconstruction code fails for unknown reason, try to add delays or random
  - Some protectors detect when you call all the redirected function one after the other
- Always prefer HBP over BP
- Prefer generic methods over signatures
  - But use signatures when it’s handy :D
- To attach your debugger to a protected process
  - Patch NtSetInformationThread before running it → bypass ThreadHideFromDebugger
  - Suspend the process → watchdog threads will be neutralized
  - In your debugger, patch DebugActiveProcess to make sure that DbgUiIssueRemoteBreakIn is not called → no thread will be created in the debugged process
Part 6: to go further...

- Armadillo
  - API redirection / IAT destruction
  - Nanomite
  - CopyMEM2
- SecuROM
  - Triggers
- ASProtect
  - Now owned by StarForce :D
  - IAT destruction, custom VM, custom anti dumps
- Themida
  - VM, anti X
- VMProtect
  - VM...
Do you have any questions?

THANK YOU FOR YOUR ATTENTION