Flare-On 10 Challenge 4: aimbot

By Christopher Gardner (@t00manybananas)

Part 1: Reversing the communication protocol

Aimbot is distributed as a simple .exe file, which when run self-describes itself as an aimbot for a game called "Sauerbraten". The form contains a single button (see Figure 1), which when clicked executes the main functionality of the program. Sauerbraten is an open source FPS game that is available from http://sauerbraten.org/.

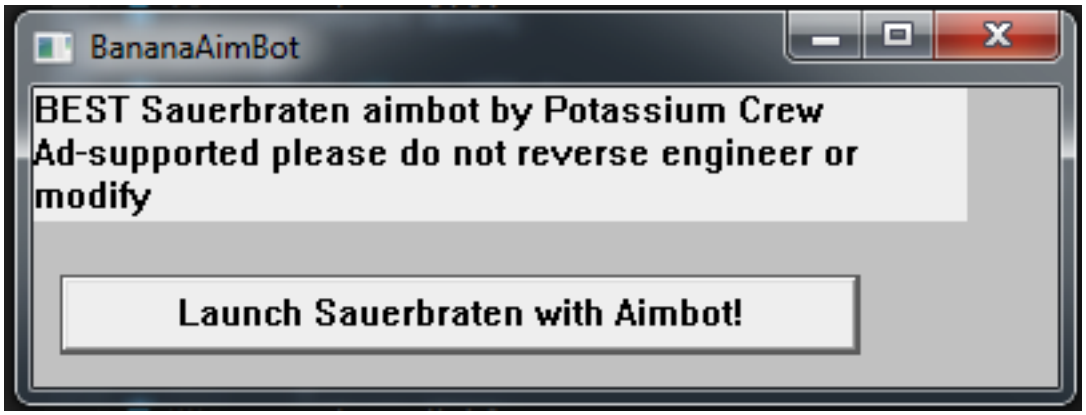

Figure 1: Aimbot GUI

The button clicked function verifies the MD5 hash of `%PROGRAMFILES(X86)%\\Sauerbraten\\bin64\\sauerbraten.exe`, then decodes several resources into `%APPDATA%\BananaBot`. It contains the following resources:

Filename	Purpose
miner.exe	XMRig coinminer
config.json	XMRig config file
aimbot.dll	Aimbot payload

Each of these resources is encrypted using AES-128 with the key `yummyvitamincjoy`. After decoding the XMRig resources, the sample launches miner.exe and sends an HTTP GET request to `http://127.0.0.1:57328/2/summary`, which the XMRig sample is configured to listen on. If the coinminer does not respond to this request, the launcher exits without dropping `aimbot.dll`.

If the GET request succeeds, the sample executes `%PROGRAMFILES(X86)%\\Sauerbraten\\bin64\\sauerbraten.exe` and injects `aimbot.dll` into the new process. The injection is achieved by writing the path of aimbot.dll into the process memory space, and then using `CreateRemoteThread` to call `LoadLibraryA` on that path in the context of the new process.

Part 2: The Aimbot DLL

When injected, the `DllMain` of `aimbot.dll` starts three threads:

- An aimbot thread, designed for Cube 2: Sauerbraten. This aimbot is fully functional but this thread is irrelevant to solving the challenge
- An anti-debug thread
- An infostealer thread, which contains the meat of the challenge

The anti-debug thread first patches the `DbgBreakPoint` function of `ntdll.dll` to return immediately. This is intended to prevent a debugger from pausing execution when attaching (it can still attach, but may fail to pause execution upon attaching). It then executes an anti-debug function in a loop, sleeping a second in between attempts and calling `ExitProcess` if the anti-debug function ever detects a debugger. This anti-debug function is also used in the infostealer thread, so understanding it is important.
The anti debug function performs the following checks, and computes a constant from some of them:

- Check `IsDebuggerPresent` and `CheckRemoteDebuggerPresent`
- Checks to see if a `ProcessDebugObject` is available for the current process
- Checks the `BeingDebugged` flag in the Process Environment Block
- Checks to see if a variety of debugger-related processes are running, and adds the number found to the constant. All but one of these are debuggers, but explorer.exe is also included in the list so the constant should be incremented by 1 if no debugger is found. The process names are hashed with a ROR hash and compared with a hardcoded list
- Adds a constant read from the memory of the parent process (ensures that the process was launched by `aimbot.exe`). This makes patching the aimbot DLL non-trivial
- Adds a constant read from the memory of the main module of the process (ensures that the DLL is running inside `sauerbraten.exe`)

The computed constant is supposed to be 0x6499f8a9. This constant is used for string decoding in the infostealer thread, so the rest of the challenge will fail to run if a debugger is detected (if it doesn't exit outright). While it is possible to solve this challenge statically, the intended solution is to correctly bypass these anti-debug checks (and the later environmental checks) to let the challenge run and output the flag. It is also possible to patch the binary to bypass some of the anti-debug checks, but some of the anti-debug checks make this harder.
The infostealer thread begins by sleeping for 5 minutes, and measures the elapsed time to ensure the sleep was successful. It then creates the directory `C:\depot`, and checks to see if XMRig is running as described earlier. However, the request must also return valid content instead of just returning an HTTP 200 status code. The challenge searches the JSON response from XMRig for the string "`version`": ", and uses that and the following bytes as an AES-128-ECB key to decrypt the next stage. The correct key is "`version`": "`6.20`. If the start of the decrypted blob is the decryption of this blob was successful, then the challenge executes the rest of the blob as shellcode.

Part 3: Infostealer shellcode blobs

The next part of the challenge is a series of chained shellcode blobs that try to steal data from a program on the system (much like an infostealer malware sample), then use some key derived from that programs data to decrypt the next blob. In essence, all the programs must be installed and setup at least somewhat properly to decrypt the next stage.

The first blob targets Steam, a gaming storefront. The challenge opens `C:\Program Files (x86)\Steam\config\config.vdf` (which should be JSON) and searches for the SentryFile key. It copies the value of that key to `C:\depot`, and then uses the first 16 bytes of the `config.vdf` file (which should be "`InstallConfigSt`) as an RC4 key for the next blob.

The next blob acts as an infostealer for Discord, a communication platform popular with gamers. It searches all files in the directory `%APPDATA%\Discord\Local Storage\leveldb` for the string `dQw4w9WgXcQ`; and if any files contain that string they are copied into `C:\depot`. If at least one file was copied, the challenge reads the first 16 bytes of `%APPDATA%\Discord\Origin Bound Certs` (which is supposed to be SQLite format 3\0) and uses that as another RC4 key for the next blob.

The final infostealer blob targets Sparrow, a cryptocurrency wallet application. It copies all .db files found in `%APPDATA%\Sparrow\wallets` to `C:\depot`, and then searches for the filename of a copied file in `%APPDATA%\Sparrow\config` (which is supposed to be JSON data). If it finds a filename as a JSON value, it extracts the key that value is stored in (which should be recentWalletFiles) and uses that as an RC4 key for the next blob.
The next blob does not target a program, but instead attempts to gather all the files stolen and exfiltrate them to a remote server. It concatenates all the files found in `C:\depot` into `C:\depot\output`, and sends that in a HTTPS POST request to `https://bighackies.flare-on.com/stolen`. It then interprets the response as four CRC32 hashes of the data it sent, looking at various offsets into the data. If all four match, the Content-Length header of the response is used as an XOR key to decrypt the next blob. Since the response should contain four CRC32 hashes (each of which is length four), the Content-Length header is expected to contain 16.
The final shellcode blob of the challenge checks a variety of constraints in the running Sauerbraten process. If all checks pass it puts the flag into memory and exits the process (via crashing it).
It first checks to see if a game is running and is playing on a map whose name starts with spcr and has one extra character in its name. There is only one map in the stock Sauerbraten install with this pattern, spcr2. It then opens the file for this map (`%%PROGRAMFILES(X86)%%\Sauerbraten\packages\base\spcr2.cfg`) and extracts bytes 0x51 to 0x5a as the first bytes of the flag (which should be computer).
The challenge then checks to see if the player has achieved 1337 kills in the game by shooting exactly 1337 bullets, and that the game has been going on for less than 5 minutes. It also checks to make sure the player is playing a local game (vs bots). This is either very difficult or impossible for a player to achieve (even with the aimbot included in the challenge), but thankfully the key bytes recovered here are easy to pull out statically.
The challenge reads some constant memory from the sauerbraten.exe process and does simple math to

calculate the remaining key bytes. After verifying that the CRC32 hash of the flag is correct it writes the flag to the heap, and exits.

The flag is: **`computer_ass1sted_ctfing@flare-on.com`**