A Developer's Introduction to Beacon Object Files

trustedsec.com/blog/a-developers-introduction-to-beacon-object-files

With the release of Cobalt Strike 4.1, a new feature has been added that allows code to be run in a more OPSEC friendly manner. This is implemented through what has been termed Beacon Object Files (BOFs). In this post, I will outline some of the less obvious restrictions of BOFs and share my workflow in an effort to assist anyone tasked with writing in this format. I will also share some code that you can reference for what finalized code might look like.

The work I completed implements some basic situational awareness commands. You can find the code for that work here https://github.com/trustedsec/CS-Situational-Awareness-BOF. The release of this code is less about these techniques themselves, and more about using them as an introduction for writing your own BOFs.

Why Should I Care About BOFs?

BOFs allow users to execute code without following Cobalt Strike's well-defined patterns. Typically, Cobalt Strike will always perform cross-process injection or start cmd.exe / powershell.exe to accomplish its more useful goals. This is well known and outlined by the creator here: https://blog.cobaltstrike.com/2017/06/23/opsec-considerations-for-beacon-commands/. When a technique is coded using a BOF, you gain the benefit of running code inside of beacon itself and without starting a child process. You also do not have the Indicators of Compromise (IoCs) associated with execute-assembly where you would start your spawn-to process and inject code into it.

At this point, you may be thinking perfect, let’s recode all of our C# tools into BOFs. This would be a mistake. BOFs are not structured in a way that allows for long running tasks. It is not a true plugin format, and it is not treated as such. The creator outlines this well here: https://www.cobaltstrike.com/help-beacon-object-files, but it is important to note.

BOFs are most appropriate for one-off commands that will return results quickly. If you want to execute a long running task in beacon, take advantage of execute-assembly and its nature of treating those items as long running jobs.

What are the Limitations of BOFs?

There are a number of constraints when working with BOFs that are not limited to their non-job nature. Below I outline all of the roadblocks I hit and how to work around them.

1. Using Global Variables

You are able to use global variables, but the loader for BOFs does not understand how to handle the virtual .bss section. This means that any global variables you define need to be initialized to a non-zero value.

2. Extended Division on x86

If you have a variable of type long long, and you are compiling for x86, this variable cannot be used in multiplication and division. Normally, the compiler accounts for this by inserting built-in function calls. The BOFs' loader will not understand these symbols. If using Visual Studio, you may have access to intrinsics that could work.

3. Crashes Calling Some Win32 APIs

If you commonly develop C code for Windows, you may be aware that Windows has multiple process heaps. I did not spend the time to identify where BOFs allocate their memory, but I did find that at least one API call (NetServerEnum) did not appreciate the heap said memory was allocated on. In these cases, call HeapAlloc (GetProcessHeap(), MEM_ZERO, size) to get memory on the standard process heap. This should fix the crash if the problem is only due to a mismatch in where the heap memory is coming from.

This is normally not required; In all the functions I touched, this was the only one with this behavior.

4. _chkstk_ms Not Found

Typically, the process stack reserves a large section of memory and commits a much smaller section. If you have a function that requires more stack space than is available, it will call a helper function that will hit a guard page and inform the operating system it needs to commit more stack memory. This is not linked in BOF. For this reason, limit the amount of stack space used in any given function to 4K or less (assumes using MinGW-w64 to compile).

5. Crash in Large Switch Statements

The BOF loader handles the code generated by small switch/case statements but not by large ones. If you have a large switch/case statement and your BOF is crashing, try switching it to if/ else if / else.

6. Relocation Truncated to Fit (Distance Between Executable Code and Other Data is >4GB)

If you are using global variables and you get this error, the workaround is declaring another global, recompile, and test. Typically, I declared 1 more non-zero initialized global and this error disappeared. This should only be an issue on x64 platforms.

What Else Should I Know About BOFs?

In the official BOF documentation, it is stated that you cannot call common functions such as strlen / “stcmp”, or anything that is not a Win32 API function. I found this to be slightly misleading. You are able to call these functions, you just need to import them from a well-defined location, such as MSVCRT, which will exist on versions of Windows all the way back to XP (and maybe even before).

The official BOF documentation states that when you copy the API definition, you need to include items such as WINBASEAPI. This is true, but it is a slight misdirection. Anything you are importing using BOF's Dynamic Function Resolution must have some form of __declspec(dllimport). This includes items from something like MSVCRT, which does not define __declspec(dllimport) in its header but is usable via Dynamic Function Resolution if you add the decorator.

What is a Good Workflow for Developing BOFs?

Many workflows can exist for developing BOFs. I have outlined mine below, which may or may not work for you.

1. Have a header file that has all of the Dynamic Function Resolution imports written—this expediates future items.
2. Include any helper functions in a .c file that you #include into your individual BOF projects. Try to keep this file small, as its object code will go into everything.
3. Use precompiler defines to allow testing your functions with and without the use of Dynamic Function Resolution.
4. Use something like https://github.com/dtmsecurity/bof_helper to assist with creating Dynamic Function Resolution imports. It will not work for everything, but it is a good first pass.
5. If step 4 fails, use the helper functions in the section below to find the header definition, adding DECLSPEC_IMPORT if required, and identify which lib/dll is needed to import and run this code.
6. Code and profit.

Helper Alias/Function

You may find these helpful when writing BOFs:

```
function getfunc {
    grep -r -I "$1" "/usr/local/Cellar/mingw-w64/7.0.0_2/toolchain-x86_64/x86_64-
w64-mingw32/include/";
}
export -f getfunc
```

The path after $1 should point to the include directory of your mingw-w64 install. Use this to get potential matches for the function header definition.

```
function boflib {
    x86_64-w64-mingw32-nm -g /usr/local/Cellar/mingw-w64/7.0.0_2/toolchain-
x86_64/mingw/lib/* --defined-only -A 2>/dev/null | grep "$1";
}
export -f boflib
```

The path after -g should point to your mingw-w64 installs lib directory. This will attempt to find potential libs that export your desired function. After finding the lib, it is extremely likely that Windows has a matching .dll that also exports this function. If this fails, perform manual verification for the .dll on a Windows target.

Summary

BOFs are a great addition to Cobalt Strike that will enhance the teams that properly leverage them. They are relatively easy to develop for once you understand some of the limitations and gotchas of this new format. At this point, I advise you to check out our implementation of some basic situational awareness commands (linked in the intro) and to continue in your quest for more advanced attacker emulation.

Help us to make our blog better!