Determining programmatically whether a file was built with LAA, ASLR, DEP, or OS-assisted /GS

devblogs.microsoft.com/oldnewthing/20150518-00

May 18, 2015

Raymond Chen

Today's Little Program parses a module to determine whether or not it was built with the following flags:

Remember, Little Programs do little error checking. In particular, this Little Program does no range checking, so a malformed binary can result in wild pointers.

```
#include <windows.h>
#include <imagehlp.h>
#include <stdio.h> // horrors! mixing stdio and C++!
#include <stddef.h>

class MappedImage
{
public:
 bool MapImage(const char* fileName);
 void ProcessResults();
 ~MappedImage();

private:
 WORD GetCharacteristics();

 template<typename T>
 WORD GetDllCharacteristics();

 template<typename T>
 bool HasSecurityCookie();

private:
 HANDLE file_ = INVALID_HANDLE_VALUE;
 HANDLE mapping_ = nullptr;
 void *imageBase_ = nullptr;
 IMAGE_NT_HEADERS* headers_ = nullptr;
 int bitness_ = 0;
};

bool MappedImage::MapImage(const char* fileName)
{
 file_ = CreateFile(fileName, GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
 if (file_ == INVALID_HANDLE_VALUE) return false;

 mapping_ = CreateFileMapping(file_, NULL, PAGE_READONLY,
                              0, 0, NULL);
 if (!mapping_) return false;

 imageBase_ = MapViewOfFile(mapping_, FILE_MAP_READ, 0, 0, 0);
 if (!imageBase_) return false;

 headers_ = ImageNtHeader(imageBase_);
 if (!headers_) return false;
 if (headers_->Signature != IMAGE_NT_SIGNATURE) return false;

 switch (headers_->OptionalHeader.Magic) {
 case IMAGE_NT_OPTIONAL_HDR32_MAGIC: bitness_ = 32; break;
```

```
 case IMAGE_NT_OPTIONAL_HDR64_MAGIC: bitness_ = 64; break;
 default: return false;
 }

 return true;
}

MappedImage::~MappedImage()
{
 if (imageBase_) UnmapViewOfFile(imageBase_);
 if (mapping_) CloseHandle(mapping_);
 if (file_ != INVALID_HANDLE_VALUE) CloseHandle(file_);
}

WORD MappedImage::GetCharacteristics()
{
 return headers_->FileHeader.Characteristics;
}

template<typename T>
WORD MappedImage::GetDllCharacteristics()
{
  return reinterpret_cast<T*>(headers_)->
    OptionalHeader.DllCharacteristics;
}

template<typename T>
bool MappedImage::HasSecurityCookie()
{
 ULONG size;
 T *data = static_cast<T*>(ImageDirectoryEntryToDataEx(
    imageBase_, TRUE, IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG,
    &size, NULL));
 if (!data) return false;
 ULONG minSize = offsetof(T, SecurityCookie) +
                 sizeof(data->SecurityCookie);
 if (size < minSize) return false;
 if (data->Size < minSize) return false;
 return data->SecurityCookie != 0;
}

void MappedImage::ProcessResults()
{
 printf("%d-bit binary\n", bitness_);
 auto Characteristics = GetCharacteristics();
 printf("Large address aware: %s\n",
    (Characteristics & IMAGE_FILE_LARGE_ADDRESS_AWARE)
    ? "Yes" : "No");

 auto DllCharacteristics = bitness_ == 32
    ? GetDllCharacteristics<IMAGE_NT_HEADERS32>()
    : GetDllCharacteristics<IMAGE_NT_HEADERS64>();
```

```
 printf("ASLR: %s\n",
    (DllCharacteristics & IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE)
    ? "Yes" : "No");
 printf("ASLR^2: %s\n",
    (DllCharacteristics & IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA)
    ? "Yes" : "No");
 printf("DEP: %s\n",
    (DllCharacteristics & IMAGE_DLLCHARACTERISTICS_NX_COMPAT)
    ? "Yes" : "No");
 printf("TS Aware: %s\n",
    (DllCharacteristics & IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE)
    ? "Yes" : "No");

 bool hasSecurityCookie =
    bitness_ == 32 ? HasSecurityCookie<IMAGE_LOAD_CONFIG_DIRECTORY32>()
                   : HasSecurityCookie<IMAGE_LOAD_CONFIG_DIRECTORY64>();
 printf("/GS: %s\n", hasSecurityCookie
    ? "Yes" : "No");
}

int __cdecl main(int argc, char**argv)
{
 MappedImage mappedImage;
 if (mappedImage.MapImage(argv[1])) {
  mappedImage.ProcessResults();
 }
 return 0;
}
```

Let's see what happened.

First we use the `MapImage` method to load the binary and map it into memory. While we're at it, we sniff at the headers to determine whether it is a 32-bit or 64-bit binary.

The `GetCharacteristics` method merely extracts the `Characteristics` from the `FileHeader`. This is easy because the `FileHeader` is the same for 32-bit and 64-bit binaries.

The `GetDllCharacteristics` method has two versions depending on the image bitness. In both cases, it extracts the `DllCharacteristics` field, but the location of the field depends on the structure.

The `HasSecurityCookie` method also has two versions depending on the image bitness. The minimum size necessary to get OS-assisted stack overflow protection is the size that encompasses the `SecurityCookie` member, and in order to get that extra protection, the member needs to be nonzero.

What is OS-assisted stack overflow protection?

First, I'm going to assume that you've read Compiler Security Checks In Depth.

Okay, welcome back.

In theory, `/GS` could be implemented entirely in application code, with no need for operating system assistance. And in fact, that's what happens when the executable is run on older versions of Windows (like Windows 98 or Windows 2000). But the module can tell the operating system, "Hey, here is where I put my security cookie," and if the operating system understands this field, then it will go in and make the security cookie even more randomer than random by mixing in some cryptographically secure random bits.

Okay, so that's the program. Note that some of these flags are meaningless in DLLs, so be careful to interpret the output correctly.

Raymond Chen

Follow