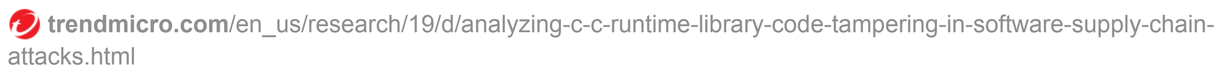
C/C++ Runtime Library Code Tampering in Supply Chain


trendmicro.com/en_us/research/19/d/analyzing-c-c-runtime-library-code-tampering-in-software-supply-chain-attacks.html

April 22, 2019

For the past few years, the security industry's very backbone — its key software and server components — has been the subject of numerous attacks through cybercriminals' various works of compromise and modifications. Such attacks involve the original software's being compromised via malicious tampering of its source code, its update server, or in some cases, both. In either case, the intention is to always get into the network or a host of a targeted entity in a highly inconspicuous fashion — which is known as a supply chain attack. Depending on the attacker's technical capabilities and stealth motivation, the methods used in the malicious modification of the compromised software vary in sophistication and astuteness.

Four major methods have been observed in the wild:

1. The injection of malicious code at the source code level of the compromised software, for native or interpreted/just-in-time compilation-based languages such as C/++, Java, and .NET.
2. The injection of malicious code inside C/C++ compiler runtime (CRT) libraries, e.g., poisoning of specific C runtime functions.

3. Other less intrusive methods, which include the compromise of the update server such that instead of deploying a benign updated version, it serves a malicious implant. This malicious implant can come from the same compromised download server or from another completely separate server that is under the attacker's control.
4. The repackaging of legitimate software with a malicious implant. Such trojanized software is either hosted on the official yet compromised website of a software company or spread via BitTorrent or other similar hosting zones.

This blog post will explore and attempt to map multiple known supply chain attack incidents that have happened in the last decade through the four methods listed above. The focus will be on Method 2, whereby a list of all poisoned C/C++ runtime functions will be provided, each mapped to its unique malware family. Furthermore, the ShadowPad incident is taken as a test case, documenting how such poisoning happens.

Methods 1 and 2 stand out from the other methods because of the nature of their operation, which is the intrusive and more subtle tampering of code — they are a category in their own right. However, Method 2 is far more insidious since any tampering in the code is not visible to the developer or any source code parser; the malicious code is introduced at the time of compilation/linking.

Examples of attacks that used a combination of Methods 1 and 3 are:

- The trojanization of MediaGet, a BitTorrent client, via a poisoned update (mid-February 2018). The change employed involved a malicious update component and a trojanized copy of the file *mediaget.exe*.
- The Nyetya/MeDoc attack on M.E.Doc, an accounting software by Intellect Service, which delivered the destructive ransomware Nyetya/NotPetya by manipulating its update system (April 2017). The change employed involved backdooring of the .NET module *ZvitPublishedObjects.dll*.
- The KingSlayer attack on EventID, which resulted in the compromise of the Windows Event Log Analyzer software's source code (service executable in .NET) and update server (March 2015).

An example of an attack that solely made use of Method 3 is the Monju incident, which involved the compromise of the update server for the media player GOM Player by GOMLab and resulted in the distribution of a variant of Gh0st RAT toward specific targets (December 2013).

For Method 4, we have the Havex incidents, which involved the compromise of multiple industrial control system (ICS) websites and software installers (different dates in 2013 and 2014).

Examples of attacks that used a combination of Methods 2 and 3 are:

- Operation ShadowHammer, which involved the compromise of a computer vendor's update server to target an unknown set of users based on their network adapters' media access control (MAC) addresses (June 2018). The change employed involved a malicious update component.
- An attack on the gaming industry (Winnti.A), which involved the compromise of three gaming companies and the backdooring of their respective main executables (publicized in March 2019).
- The CCleaner case, which involved the compromise of Piriform, resulting in the backdooring of the CCleaner software (August 2017).
- The ShadowPad case, which involved the compromise of NetSarang Computer, Inc., resulting in the backdooring of all of the company's products (July 2017). The change employed involved malicious code that was injected into the library *nssock2.dll*, which was used by all of the company's products.

Methods 2 and 3 were also used by the Winnti group, which targeted the online video game industry, compromising multiple companies' update servers in an attempt to spread malicious implants or libraries using the AheadLib tool (2011).

Another example is the XcodeGhost incident (September 2015), in which Apple's Xcode integrated development environment (IDE) and the compiler's CoreServices Mach-O object file were modified to include malware that would infect every iOS app built (via the linker) with the trojanized Xcode IDE. The trojanized version was hosted on multiple Chinese file sharing services, resulting in hundreds of trojanized apps' landing on the iOS App Store unfettered.

An interesting case that shows a different side to the supply chain attack methods is the event-stream incident (November 2018). Event-stream is one of the widely used packages by npm (Node.js package manager), a package manager for the JavaScript programming language. A package known as flatmap-stream was added as a direct dependency to the event-stream package. The original author/maintainer of the event-stream package delegated publishing rights to another person, who then added the malicious flatmap-stream package. This malicious package targeted specific developers working on the release build scripts of the bitcoin wallet app Copay, all for the purpose of stealing bitcoins. The malicious code got written into the app when the build scripts were executed, thereby adding another layer of covertness.

In most supply chain attack cases that have been happening for almost a decade, the initial infection vector is unknown or at least not publicly documented. Moreover, the particulars of how the malicious code gets injected into the benign software codebase are not documented either, whether from a forensics or a tactics, techniques, and procedures (TTP) standpoint. However, we will attempt to show how Method 2, which employs sophisticated tampering of code and is harder to detect, is used by attackers in a supply chain attack, using the ShadowPad case as our sample for analysis.

An In-Depth Analysis of Method 2 – Case Study: ShadowPad

There are subtle differences and observations between tampering with the original source code, as in Method 1, and tampering with the C/C++ runtime libraries, as in Method 2. Depending on the nature and location of the changes, the former might be easier to spot, whereas the latter would be much harder to detect if no file monitoring and integrity checks had been in place.

All of the reported cases where the C/C++ runtime time libraries are poisoned or modified are for Windows binaries. Each case has been statically compiled with the Microsoft Visual C/C++ compiler with varying linker versions. Additionally, all of the poisoned functions are not part of the actual C/C++ standard libraries, but are specific to Microsoft Visual C/C++ compiler runtime initialization routines. Table 1 shows the list of all known malware families with their tampered runtime functions.

Malware Family	Poisoned Microsoft Visual C/C++ Runtime Functions
ShadowHammer	__crtExitProcess(UINT uExitCode) // exits the process. Checks if it's part of a managed app // it is a CRT wrapper for ExitProcess
Gaming industry (HackedApp.Winnti.A)	__scrt_common_main_seh(void) *// entrypoint of the c runtime library (_mainCRTStartup) with support for structured exception handling which calls the program's main() function*
CCleaner	*Stage 1:* __scrt_common_main_seh(void)
Stage 2 -> dropped(32- bit)	_security_init_cookie()
Stage 2 -> dropped (64- bit)	_security_init_cookie()
void __security_init_cookie(void); *// Initializes the global security cookie // used for buffer overflow protection*	
ShadowPad	_initterm(_PVFV * pfbegin, _PVFV * pfend); *// call entries in function pointer table // The entry (0x1000E600) is the malicious one*

Table 1. List of poisoned/modified Microsoft Visual CRT functions in supply chain attacks

It's the linker's responsibility to include the necessary CRT library for providing the startup code. However, a different CRT library could be specified via an explicit linker flag. Otherwise, the default statically linked CRT library *libcmt.lib*, or another, is used. The startup

code performs various environment setup operations prior to executing the program's *main()* function. Such operations include exception handling, thread data initialization, program termination, and cookie initialization. It's important to note that the CRT implementation is compiler-, compiler option-, compiler version-, and platform-specific.

Microsoft used to ship the Visual C runtime library headers and compilation files that developers could build themselves. For example, for Visual Studio 2010, such headers would exist under "Microsoft Visual Studio 10.0\VC\crt", and the actual implementation of the ShadowPad poisoned function *_initterm()* would reside inside the file *crt0dat.c* as follows (all comments were omitted for readability purposes):

```
void __cdecl _initterm (_PVFV * pfbegin, _PVFV * pfend)
{
    while ( pfbegin < pfend )
    {
        if ( *pfbegin != NULL )
            (**pfbegin)();
        ++pfbegin;
    }
}
```

This internal function is responsible for walking a table of function pointers (skipping null entries) and initializing them. It's called only during the initialization of a C++ program. The poisoned DLL *nssock2.dll* is written in the C++ language.

The argument *pfbegin* points to the first valid entry on the table, while *pfend* points to the last valid entry. The definition of the function type *_PVFV* is inside the CRT file *internal.h*:

```
typedef void (__cdecl *_PVFV)(void);
```

The above function is defined in the *crt0dat.c* file. The object file *crt0dat.obj* resides inside the library file *libcmt.lib*.

Figure 1 shows ShadowPad's implementation of *_initterm()*.

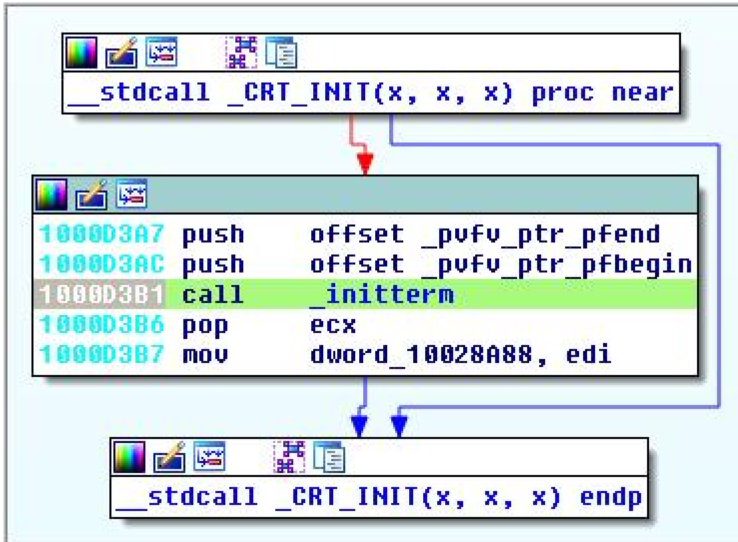


Figure 1. ShadowPad poisoned *_initterm()* runtime function

Figure 2 shows the function pointer table for ShadowPad's *_initterm()* function as pointed to by *pfbegin* and *pfend*. This table is used for constructing objects at the beginning of the program particularly for calling C++ constructors, which is what's happening in the screenshot below.

```
.rdata:1000F698 _pvfv_ptr_pfbegin db    0             ; DATA XREF: _CRT_INIT(x,x,x)+1A6↑o
.rdata:1000F699                   db    0
.rdata:1000F69A                   db    0
.rdata:1000F69B                   db    0
.rdata:1000F69C                   dd offset `dynamic initializer for 'afxModuleState''(void)
.rdata:1000F6A0                   dd offset malicious_code
.rdata:1000F6A4                   dd offset sub_1000E510
.rdata:1000F6A8                   dd offset sub_1000E530
.rdata:1000F6AC                   dd offset sub_1000E550
.rdata:1000F6B0                   dd offset sub_1000E570
.rdata:1000F6B4                   dd offset sub_1000E590
.rdata:1000F6B8                   dd offset sub_1000E5B0
.rdata:1000F6BC                   dd offset sub_1000E5C0
.rdata:1000F6C0                   dd offset sub_1000E5D0
.rdata:1000F6C4                   dd offset sub_1000E5E0
.rdata:1000F6C8                   dd offset sub_1000E5F0
.rdata:1000F6CC _pvfv_ptr_pfend   db    0             ; DATA XREF: _CRT_INIT(x,x,x)+1A1↑o
.rdata:1000F6CD                   db    0
.rdata:1000F6CE                   db    0
.rdata:1000F6CF                   db    0
```

Figure 2. Function pointer table for ShadowPad poisoned *_initterm()* runtime function

As shown in Figure 2, the function pointer entry labeled *malicious_code* at the virtual address 0x1000F6A0 has been poisoned to point to a malicious code (0x1000E600). It's more accurate to say that it is the function pointer table that was poisoned rather than the function *_initterm()*.

Figure 3 shows the cross-reference graph of the *_initterm()* CRT function as referenced by the compiled ShadowPad code. The graph shows all call paths (reachability) that lead to it, and all other calls it makes itself. The actual call path that leads to executing the ShadowPad

code is:

DllEntryPoint() -> __DllmainCRTStartup() -> _CRT_INIT() -> _initterm() -> __imp_initterm() -> malicious_code() via function pointer table.

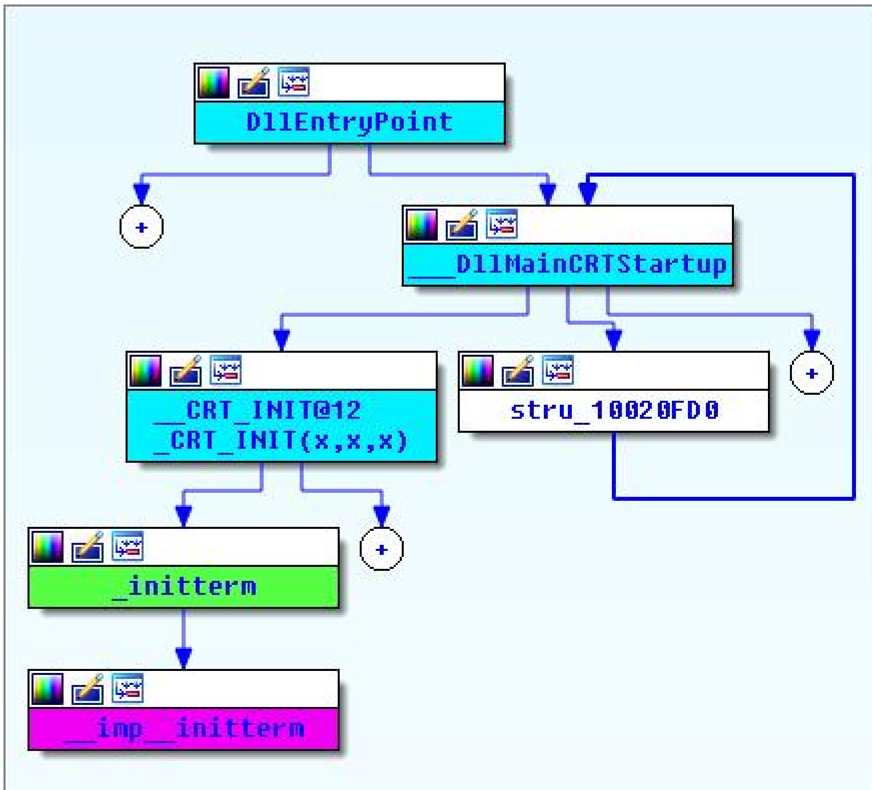


Figure 3. Call cross-reference graph for ShadowPad poisoned _*initterm()* runtime function

Note that the internal function _*initterm()* is called from within the CRT initialization function __*CRT_INIT()*, which is responsible for C++ DLL initialization and has the following prototype:

```
BOOL WINAPI _CRT_INIT(hDllHandle, dwReason, lpreserved)
```

One of its responsibilities is invoking the C++ constructors for the C++ code in the DLL *nssock2.dll*, as demonstrated earlier. The said function is implemented inside the CRT file *crtdll.c* -> object file *crtdll.obj* -> library file *msvcrt.lib*.

The following code snippet shows the actual implementation of the function _*CRT_INIT()*.

```
BOOL WINAPI _CRT_INIT(HANDLE hDllHandle, DWORD dwReason, LPVOID lpreserved) {
  /* truncated for readability... */

  if (__native_startup_state != __uninitialized) {
    _amsg_exit(_RT_CRT_INIT_CONFLICT);
  } else {

    /* Set the native startup state to initializing */

    __native_startup_state = __initializing;

    /* Invoke C initializers */

    #ifndef _SYSCRT
    if (_initterm_e(__xi_a, __xi_z) != 0)
      return FALSE;
    #else /* _SYSCRT */
      _initterm((_PVFV * )(void * ) __xi_a, (_PVFV * )(void * ) __xi_z);
    #endif /* _SYSCRT */

     /* Invoke C++ constructors */
    _initterm(__xc_a, __xc_z);

     /* Set the native initialization state to initialized */
    __native_startup_state = __initialized;
  }

  /* truncated for readability... */
  return TRUE;
}
```

So, how could an attacker poison any of those CRT functions? It's possible to overwrite the original benign *libcmt.lib*/*msvcrt.lib* library with a malicious one, or modify the linker flag such that it points to a malicious library file. Another possibility is by hijacking the linking process such that as the linker is resolving all references to various functions, the attacker's tool monitors this process, intercepts it, and feeds it a poisoned function definition instead. The backdooring of the compiler's key executables, such as the linker binary itself, can be another stealthy poisoning vector.

Conclusion

Although the attacks for Method 2 are very low in number, difficult to predict, and possibly targeted, when one takes place, it can be likened to a black swan event: It will catch victims off guard and its impact will be widespread and catastrophic.

Tampering with CRT library functions in supply chain attacks is a real threat that requires further attention from the security community, especially when it comes to the verification and validation of the integrity of development and build environments.

Steps could be taken to ensure clean software development and build environments. Maintaining and cross-validating the integrity of the source code and all compiler libraries and binaries are good starting points. The use of third-party libraries and code must be vetted and scanned for any malicious indicators prior to integration and deployment. Proper network segmentation is also essential for separating critical assets in the build and distribution (update servers) environments from the rest of the network. Important as well is the enforcement of very strict access with multifactor authentication to the release build servers and endpoints. Of course, these steps do not exclude or relinquish the developers themselves from the responsibility of continuously monitoring the security of their systems.