

## DYLIB HIJACKING ON OS X

*Patrick Wardle*  
Synack, USA

(This paper was presented at CanSecWest 2015.)

DLL hijacking is a well known class of attack which was always believed only to affect the *Windows* OS. However, this paper will show that *OS X* is similarly vulnerable to dynamic library hijacks. By abusing various features and undocumented aspects of *OS X*'s dynamic loader, attackers need only to 'plant' specially crafted dynamic libraries to have malicious code automatically loaded into vulnerable applications. Using this method, such attackers can perform a wide range of malicious and subversive actions, including stealthy persistence, load-time process injection, security software circumvention, and a *Gatekeeper* bypass (affording opportunities for remote infection). Since this attack abuses legitimate functionality of the OS, it is challenging to prevent and unlikely to be patched. However, this paper will present techniques and tools that can uncover vulnerable binaries as well as detect if a hijacking has occurred.

## BACKGROUND

Before detailing the dynamic library (dylib) hijacking attack on *OS X*, dynamic link library (DLL) hijacking on *Windows* will briefly be reviewed. As the two attacks are conceptually quite similar, examining the well-understood *Windows* attack can help in gaining an understanding of the former.

DLL hijacking on *Windows* is best explained by *Microsoft*:

'When an application dynamically loads a dynamic link library (DLL) without specifying a fully qualified path name, Windows tries to locate the DLL by searching a well-defined set of directories. If an attacker gains control of one of the directories, they can force the application to load a malicious copy of the DLL instead of the DLL that it was expecting.' [1]

To reiterate, the default search behaviour of the *Windows* loader is to search various directories (such as the application's directory or the current working directory) before the *Windows* system directory. This can be problematic if an application attempts to load a system library via an insufficiently qualified path (i.e. just by its name). In such a scenario, an attacker may 'plant' a malicious DLL (the name of which matches that of the legitimate system DLL) in one of the primary search directories. With this malicious DLL in place, the *Windows* loader will find the

attacker's library before the legitimate DLL and blindly load it into the context of the vulnerable application.

This is illustrated in Figures 1 and 2, where a vulnerable application (Figure 1) is hijacked by a malicious DLL that has been planted in the primary search directory (Figure 2).

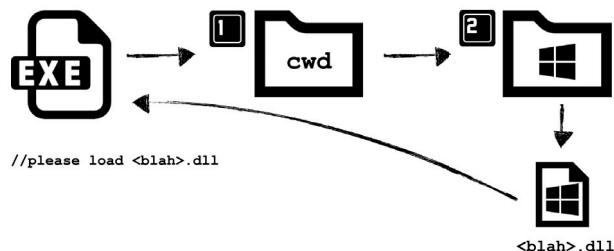


Figure 1: Loading the legitimate system DLL.

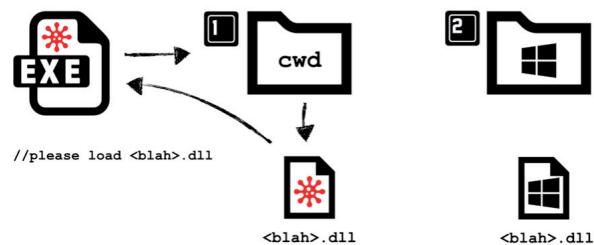


Figure 2: Loading the attacker's malicious DLL.

DLL hijacking attacks initially gained notoriety in 2010 and quickly grabbed the attention of both the media and malicious attackers. Also known as 'binary planting', 'insecure library loading' or 'DLL preloading', the discovery of this vulnerability is often attributed to H.D. Moore [2, 3]. However, the NSA was actually the first to note this flaw, 12 years prior to Moore, in 1998. In the NSA's unclassified 'Windows NT Security Guidelines', the organization both describes and warns of DLL hijacking:

'It is important that penetrators can't insert a "fake" DLL in one of these directories where the search finds it before a legitimate DLL of the same name.' [4]

To an attacker, DLL hijacking affords many useful scenarios. For example, such attacks can allow a malicious library to stealthily be persisted (without modifying the registry or other components of the OS), privileges to be escalated, and even provides the means for remote infection.

Malware authors were fairly quick to realize the benefits of DLL hijacking. In a blog post entitled 'What the fxsst?' [5], *Mandiant* researchers described how they had uncovered various unrelated malware samples all named 'fxsst.dll'.

```

//paths to abuse
char* uacTargetDir[] = {"system32\sysprep", "ehome"};
char* uacTargetApp[] = {"sysprep.exe", "mcx2prov.exe"};
char* uacTargetDll[] = {"cryptbase.dll", "CRYPTSP.dll"};

//execute vulnerable application & perform DLL hijacking attack
if(Exec(&exitCode, "cmd.exe /C %s", targetPath))
{
    if(exitCode == UAC_BYPASS_MAGIC_RETURN_CODE)
        DBG("UAC BYPASS SUCCESS")
...

```

Figure 3: Carberp abusing a DLL hijack to bypass UAC.

Upon closer inspection, they found that the samples were all exploiting a DLL hijacking vulnerability in the *Windows* shell (Explorer.exe), that provided a stealthy method of persistence. Specifically, as Explorer.exe was installed in C:\Windows, planting a library named fxsst.dll in the same directory would result in the persistence of the malicious DLL as the loader searched the application's directory before the system directory where the legitimate fxsst.dll lived.

Another example of malware using a DLL hijack can be found within the leaked source code for the banking trojan 'Carberp' [6]. The source code shows the malware bypassing User Account Control (UAC) via a DLL hijack of sysprep.exe (see Figure 3). This binary is an auto-elevated process, meaning that it requires no UAC prompt to gain elevated status. Unfortunately, it was found to be vulnerable to a DLL hijacking attack and would load a maliciously planted DLL (named cryptbase.dll) into its elevated process context [7].

These days, DLL hijacking on *Windows* is somewhat uncommon. Microsoft was swift to respond to attacks, patching vulnerable applications and detailing how others could avoid this issue (i.e. simply by specifying an absolute, or fully qualified path for imported DLLs) [8]. Moreover, OS-level mitigations were introduced, which if enabled via the SafeDllSearchMode and/or CWDIllegalInDllSearch registry keys, stop the majority of DLL hijackings generically.

## DYLIB HIJACKING ON OS X

It has always been assumed that dynamic library hijacking was a *Windows*-only problem. However, as one astute *StackOverflow* user pointed out in 2010, 'any OS which allows for dynamic linking of external libraries is theoretically vulnerable to this' [9]. It took until 2015 for him to be proved correct – this paper will reveal an equally devastating dynamic library hijack attack affecting *OS X*.

The goal of the research presented here was to determine whether *OS X* was vulnerable to a dynamic library attack.

Specifically, the research sought to answer the question: could an attacker plant a malicious *OS X* dynamic library (dylib) such that the OS's dynamic loader would load it automatically into a vulnerable application? It was hypothesized that, much like DLL hijacking on *Windows*, such an attack on *OS X* would provide an attacker with a myriad of subversive capabilities. For example, stealthy persistence, load-time process injection, security software circumvention, and perhaps even 'remote' infection.

It should be noted that several constraints were placed upon this undertaking. First, success was constrained by disallowing any modification to the system – except for the creation of files (and if necessary folders). In other words, the research ignored attack scenarios that required the subverting of existing binaries (e.g. patching) or modifications to existing OS configuration files (e.g. 'auto-run' plists, etc.). As such attacks are well known and trivial both to prevent and to detect, they were ignored. The research also sought a method of hijack that was completely independent of the user's environment. *OS X* provides various legitimate means to control the environment in a manner that could coerce the loader to load malicious libraries automatically into a target process. These methods, such as setting the DYLD\_INSERT\_LIBRARIES environment variable, are user-specific and, again, well known and easy to detect. As such, they were of little interest and were ignored.

The research began with an analysis of the *OS X* dynamic linker and loader, dyld. This binary, found within /usr/bin, provides standard loader and linker functionality including finding, loading and linking dynamic libraries.

As Apple has made dyld open source [10], analysis was fairly straightforward. For example, reading the source code provided a decent understanding of dyld's actions as an executable is loaded and its dependent libraries are loaded and linked in. The following briefly summarizes the initial steps taken by dyld (focusing on those that are relevant to the attack described in this paper):

1. As any new process is started, the kernel sets the user-mode entry point to point to `_dyld_start` (`dyldStartup.s`). This function simply sets up the stack then jumps to `dyldbootstrap::start()`, which in turn calls the loader's `_main()`.
2. Dyld's `_main()` function (`dyld.cpp`) invokes `link()`, which then calls an `ImageLoader` object's `link()` method to kick off the linking process for the main executable.
3. The `ImageLoader` class (`ImageLoader.cpp`) exposes many functions that dyld calls in order to perform various binary image loading logic. For example, the class contains a `link()` method. When called, this invokes the object's `recursiveLoadLibraries()` method to perform the loading of all dependent dynamic libraries.
4. The `ImageLoader`'s `recursiveLoadLibraries()` method determines all required libraries and invokes the `context.loadLibrary()` function on each. The `context` object is simply a structure of function pointers that is passed around between methods and functions. The `loadLibrary` member of this structure is initialized with the `libraryLocator()` function (`dyld.cpp`), which simply calls the `load()` function.
5. The `load()` function (`dyld.cpp`) calls various helper functions within the same file, named `loadPhase0()` through to `loadPhase5()`. Each function is responsible for handling a specific task of the load process, such as resolving paths or dealing with environment variables that can affect the load process.
6. After `loadPhase5()`, the `loadPhase6()` function finally loads (maps) the required dylibs from the file system into memory. It then calls into an instance of the `ImageLoaderMachO` class in order to perform Mach-O-specific loading and linking logic on each dylib.

With a basic understanding of dyld's initial loading logic, the research turned to hunting for logic that could be abused to perform a dylib hijack. Specifically, the research was interested in code in the loader that didn't error out if a dylib wasn't found, or code that looked for dylibs in multiple locations. If either of these scenarios was realized within the loader, it was hoped that an OS X dylib hijack could be performed.

The initial scenario was investigated first. In this case, it was hypothesized that if the loader could handle situations where a dylib was not found, an attacker (who could identify such situations) could place a malicious dylib in this presumed location. From then on, the loader would now 'find' the planted dylib and blindly load the attacker's malicious code.

Recall that the loader calls the `ImageLoader` class's `recursiveLoadLibraries()` method to both find and load all required libraries. As shown in Figure 4, the loading code

is wrapped in a try/catch block to detect dylibs that fail to load.

```
//attempt to load all required dylibs
void ImageLoader::recursiveLoadLibraries( ... ) {
    //get list of libraries this image needs
    DependentLibraryInfo libraryInfos[fLibraryCount];
    this->doGetDependentLibraries(libraryInfos);

    //attempt to load each dylib
    for(unsigned int i=0; i < fLibraryCount; ++i) {
        //load
        try {
            dependentLib = context.loadLibrary(libraryInfos[i], ... );
            ...
        } catch(const char* msg) {
            if(requiredLibInfo.required)
                throw dyld::mkStringf("Library not loaded: %s\n"
                                      "Referenced from: %s\n"
                                      "Reason: %s",
                                      requiredLibInfo.name, this->getRealPath(), msg);

            //ok if weak library not found
            dependentLib = NULL;
        }
    }
}
```

Figure 4: Error logic for dylib load failures.

Unsurprisingly, there is logic to throw an exception (with a message) if a library fails to load. Interestingly though, this exception is only thrown if a variable named 'required' is set to true. Moreover, the comment in the source code indicates that failure to load 'weak' libraries is OK. This seems to indicate that some scenario exists where the loader is OK with missing libraries – perfect!

Digging deeper into the loader's source code revealed where this 'required' variable is set. Specifically, the `doGetDependentLibraries()` method of the `ImageLoaderMacho` class parses the load commands (described below) and sets the variable based on whether or not the load command is of type `LC_LOAD_WEAK_DYLIB`.

Load commands are an integral component of the Mach-O file format (OS X's native binary file format). Embedded immediately following the Mach-O header, they provide various commands to the loader. For example, there are load commands to specify the memory layout of the binary, the initial execution state of the main thread, and information about the dependent dynamic libraries for the binary. To view the load commands of a compiled binary, a tool such as `MachOView` [11] or `/usr/bin/otool` (with the `-l` command-line flag) can be used (see Figure 6).

The code in Figure 5 shows the loader iterating over all the load commands within a binary, looking for those that specify a dylib import. The format of such load commands (e.g. `LC_LOAD_DYLIB`, `LC_LOAD_WEAK_DYLIB`, etc.) can be found in the `mach-o/loader.h` file.

```

//get all libraries required by the image
void ImageLoaderMachO::doGetDependentLibraries(DependentLibraryInfo libs[]){
    //get list of libraries this image needs
    const uint32_t cmd_count = ((macho_header*)fMachOData)->ncmds;
    const struct load_command* const cmds =
        (struct load_command*)&fMachOData[sizeof(macho_header)];
    const struct load_command* cmd = cmds;

    //iterate over all load commands
    for (uint32_t i = 0; i < cmd_count; ++i){
        switch (cmd->cmd) {
            case LC_LOAD_DYLIB:
            case LC_LOAD_WEAK_DYLIB:
                ...

                //set required variable
                (&libs[index++])->required = (cmd->cmd != LC_LOAD_WEAK_DYLIB);

                break;
        }

        //go to next load command
        cmd = (const struct load_command*)((char*)cmd)+cmd->cmdsize;
    }
}

```

Figure 5: Setting the ‘required’ variable (src file?).

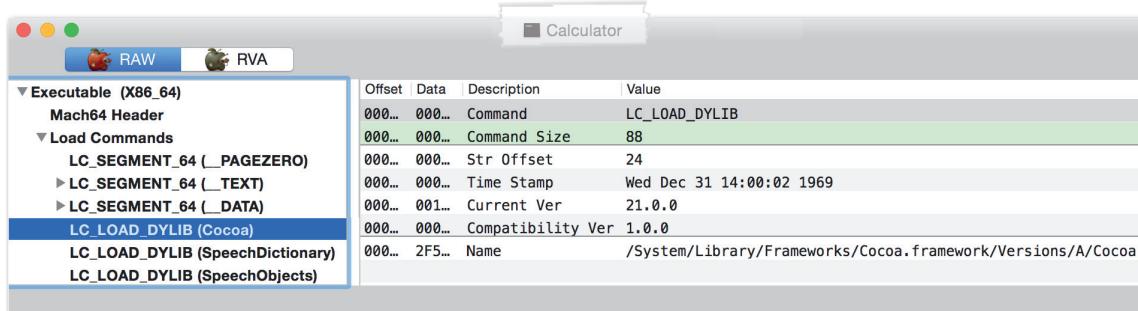


Figure 6: Dumping Calculator.app’s load commands with MachOView.

```

struct dylib_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    struct dylib dylib;
};

struct dylib
{
    union lc_str name;
    uint32_t timestamp;
    uint32_t current_version;
    uint32_t compatibility_version;
};

```

Figure 7: The format of the LC\_LOAD\_\* load commands.

For each dylib that an executable was dynamically linked against, it will contain an LC\_LOAD\_\* (LC\_LOAD\_DYLIB, LC\_LOAD\_WEAK\_DYLIB, etc.) load command. As the loader code in Figures 4 and 5 illustrates, LC\_LOAD\_DYLIB load commands specify a required dylib, while libraries

imported via LC\_LOAD\_WEAK\_DYLIB are optional (i.e. ‘weak’). In the case of the former (LC\_LOAD\_DYLIB), an exception will be thrown if the required dylib is not found, causing the loader to abort and terminate the process. However, in the latter case (LC\_LOAD\_WEAK\_DYLIB), the dylib is optional. If such a ‘weak’ dylib is not found, no harm is done, and the main binary will still be able to execute.

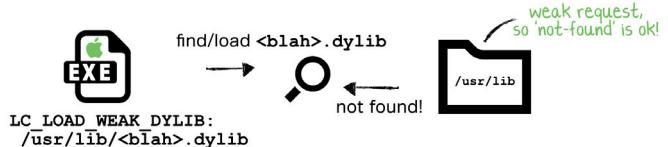


Figure 8: Attempting to load a ‘weak’ dylib (LC\_LOAD\_WEAK\_DYLIB).

This loader logic fulfilled the first hypothetical hijack scenario, and as such, provided a dylib hijack attack on OS X. Namely, as illustrated in Figure 9, if a binary specifies a weak

import that is not found, an attacker can place a malicious dylib in this presumed location. From then on, the loader will ‘find’ the attacker’s dylib and blindly load this malicious code into the process space of the vulnerable binary.

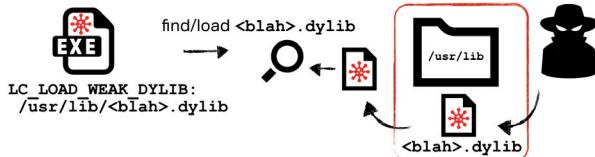


Figure 9: Hijacking an application via a malicious ‘weak’ dylib.

Recall that another hijack attack was hypothesized if a scenario existed where the loader searched for dynamic libraries in multiple locations. In this case, it was thought that an attacker would be able to place a malicious dylib in one of the primary search directories (if the legitimate dylib was found elsewhere). It was hoped that the loader would then find the attacker’s malicious dylib first (before the legitimate one), and thus naively load the attacker’s malicious library.

On OS X, load commands such as LC\_LOAD\_DYLIB always specify a path to the dynamic library (as opposed to Windows, where just the name of the library may be provided). Because a path is provided, dyld generally does not need to search various directories to find the dynamic library. Instead, it can simply go directly to the specified directory and load the dylib. However, analysis of dyld’s source code uncovered a scenario in which this generality did not hold.

Looking at the loadPhase3() function in dyld.cpp revealed some interesting logic, as shown in Figure 10.

```
//substitute @rpath with all -rpath paths up the load chain
for(const ImageLoader::RPathChain* rp=context.rpath; rp != NULL; rp=rp->next)
{
    //try each rpath
    for(std::vector<const char*>::iterator it=rp->paths->begin();
        it != rp->paths->end(); ++it){
        //build full path from current rpath
        char newPath[strlen(*it) + strlen(trailingPath)+2];
        strcpy(newPath, *it);
        strcat(newPath, "/");
        strcat(newPath, trailingPath);

        //TRY TO LOAD
        // -if this fails, will attempt next variation!
        image = loadPhase4(newPath, orgPath, context, exceptions);
        if(image != NULL)
            dyld::log("RPATH successful expansion of %s to: %s\n", orgPath,
                     newPath);
        else
            dyld::log("RPATH failed to expanding %s to: %s\n", orgPath,
                     newPath);

        //if found/load image, return it
        if(image != NULL)
            return image;
    }
}
```

Figure 10: Loading ‘rpath’ dependent libraries.

Dyld will iterate over an rp->paths vector, dynamically building paths (held within the ‘newPath’ variable) which are then loaded via the loadPhase4() function. While this does seem to fulfil the requirement of the second hijack scenario (i.e. dyld looking in multiple locations for the same dylib), a closer examination was required.

The comment on the first line of dyld’s source in Figure 10 mentions the term ‘@rpath.’ According to Apple documentation, this is a special loader keyword (introduced in OS X 10.5, Leopard) that identifies a dynamic library as a ‘run-path-dependent library’ [12]. Apple explains that a run-path dependent library ‘is a dependent library whose complete install name (path) is not known when the library is created’ [12]. Other online documentation such as [13] and [14] provides more detail, describing the role of these libraries and explaining how the @rpath keyword enables: ‘frameworks and dynamic libraries to finally be built only once and be used for both system-wide installation and embedding without changes to their install names, and allowing applications to provide alternate locations for a given library, or even override the location specified for a deeply embedded library’ [14].

While this feature allows software developers to deploy complex applications more easily, it can also be abused to perform a dylib hijack. This is true since in order to make use of run-path-dependent libraries, ‘an executable provides a list of run-path search paths, which the dynamic loader traverses at load time to find the libraries’ [12]. This is realized in code in various places within dyld, including the code snippet that was presented in Figure 10.

Since run-path dependent libraries are relatively novel and somewhat unknown, it seemed prudent to provide an example of building both a legitimate run-path-dependent library and a sample application that links against it.

A run-path-dependent library is a normal dylib whose install name is prefixed with ‘@rpath’. To create such a library in Xcode one can simply set the dylib’s installation directory to ‘@rpath’, as shown in Figure 11.



Figure 11: Building a run-path-dependent library.

Once the run-path-dependent library was compiled, examination of the LC\_ID\_DYLIB load command (which contains identifying information about the dylib) showed the

run-path of the dylib. Specifically, the ‘name’ (path) within the LC\_ID\_DYLIB load command contained the dylib’s bundle (rpathLib.framework/Versions/A/rpathLib), prefixed with the ‘@rpath’ keyword (see Figure 12).

```
$ otool -l rpathLib.framework/Versions/A/rpathLib
Load command 3
    cmd LC_ID_DYLIB
    cmdsize 72
        name @rpath/rpathLib.framework/Versions/A/rpathLib
time stamp 1 Wed Dec 31 14:00:01 1969
        current version 1.0.0
compatibility version 1.0.0
```

Figure 12: ‘@rpath’ embedded in the dylib’s ‘install name’ (path).

Building an application that linked against a run-path-dependent library was fairly straightforward as well. First, the run-path-dependent library was added to the ‘Link Binary With Libraries’ list in Xcode. Then a list of run-path search directories was added to the ‘Runpath Search Paths’ list. As will be shown, these search directories are traversed by the dynamic loader at load time in order to locate the run-path-dependent libraries.

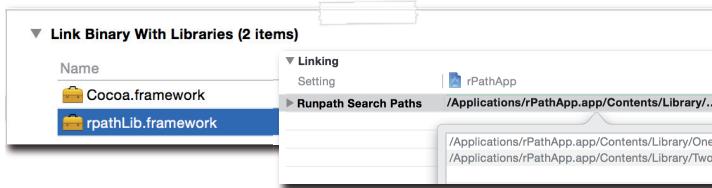


Figure 13: Linking in a @rpath’d dylib and specifying the run path search paths.

Once the application was built, dumping its load commands revealed various commands associated with the run-path library dependency. A standard LC\_LOAD\_DYLIB load command was present for the dependency on the run-path-dependent dylib, as shown in Figure 14.

```
$ otool -l rPathApp.app/Contents/MacOS/rPathApp
Load command 12
    cmd LC_LOAD_DYLIB
    cmdsize 72
        name @rpath/rpathLib.framework/Versions/A/rpathLib
time stamp 2 Wed Dec 31 14:00:02 1969
        current version 1.0.0
compatibility version 1.0.0
```

Figure 14: The dependency on the @rpath’d dylib.

In Figure 14, note that the install name (i.e. path) to the run-path-dependent dylib is prefixed with ‘@rpath’ and matches the name value from the LC\_ID\_DYLIB load command of the run-path-dependent dylib (see Figure 12). This application’s embedded LC\_LOAD\_DYLIB load

command with the run-path-dependent dylib tells the loader, ‘I depend on the rpathLib dylib, but when built, I didn’t know exactly where it would be installed. Please use my embedded run-path search paths to find it and load it!’

The run-path search paths that were entered into the ‘Runpath Search Paths’ list in Xcode generated LC\_RPATH load commands – one for each search directory. Dumping the load commands of the compiled application revealed the embedded LC\_RPATH load commands, as shown in Figure 15.

```
$ otool -l rPathApp.app/Contents/MacOS/rPathApp
Load command 18
    cmd LC_RPATH
    cmdsize 64
        path /Applications/rPathApp.app/Contents/Library/One
Load command 19
    cmd LC_RPATH
    cmdsize 64
        path /Applications/rPathApp.app/Contents/Library/Two
```

Figure 15: The embedded run-path search paths (directories).

With a practical understanding of run-path-dependent dylibs and an application that linked against one, it was easy to understand dyld’s source code which was responsible for handling this scenario at load time.

When an application is launched, dyld will parse the application’s LC\_LOAD\_\* load commands in order to load and link all dependent dylibs. To handle run-path-dependent libraries, dyld performs two distinct steps: it extracts all embedded run-path search paths and then uses this list to find and load all run-path-dependent libraries.

In order to extract all embedded run-path search paths, dyld invokes the getRPaths() method of the ImageLoader class. This method (invoked by the recursiveLoadLibraries() method) simply parses the application for all LC\_RPATH load commands. For each such load command, it extracts the run-path search path and appends it to a vector (i.e. a list), as shown in Figure 16.

```
void ImageLoaderMachO::getRPaths(..., std::vector<const char*>& paths){
    //iterate over all load commands
    // ->look for LC_RPATH and save their path's
    for(uint32_t i = 0; i < cmd_count; ++i){
        switch(cmd->cmd){
            case LC_RPATH:
                //save 'run-path' search path
                paths.push_back((char*)cmd + ((struct rpath_command*)
                    cmd)->path.offset);
                ...
            ...
        }
    }
    //keep scanning load commands...
    cmd = (const struct load_command*)((char*)cmd+cmd->cmdsize);
```

Figure 16: Extracting and saving all embedded run-path search paths.

With a list of all embedded run-path search paths, dyld can now ‘resolve’ all dependent run-path-dependent libraries. This logic is performed in the loadPhase3() function in dyld.cpp. Specifically, the code (shown in Figure 17) checks to see if a dependent library’s name (path) is prefixed with the ‘@rpath’ keyword. If so, it iterates over the list of extracted run-path search paths, replacing the ‘@rpath’ keyword in the import with the current search path. Then it attempts to load the dylib from this newly resolved directory.

```
//expand '@rpaths'
static ImageLoader* loadPhase3(...) {

//replace '@rpath' with all resolved run-path search paths & try load
else if(context.implicitRPath || (strncmp(path, "@rpath/", 7) == 0) ) {

    //get part of path after "@rpath/"
    char* trailingPath = (strncmp(path, "@rpath/", 7) == 0) ? &path[7] : path;

    //substitute @rpath with all -rpath paths up the load chain
    for(std::vector<const char*>::iterator it=rp->paths->begin();
        it != rp->paths->end(); ++it){

        //build full path from current rpath
        char newPath[strlen(*it) + strlen(trailingPath)+2];
        strcpy(newPath, *it);
        strcat(newPath, "/");
        strcat(newPath, trailingPath);

        //TRY TO LOAD
        image = loadPhase4(newPath, orgPath, context, exceptions);

        //if found/loaded image, return it
        if(image != NULL)
            return image;
    }

} //try all run-path search paths
```

Figure 17: Searching run-path search directories for @rpath’d dylibs.

It is important to note that the order of the directories that dyld searches is deterministic and matches the order of the embedded LC\_RPATH load commands. Also, as is shown in the code snippet in Figure 17, the search continues until the dependent dylib is found or all paths have been exhausted.

Figure 18 illustrates this search conceptually. The loader (dyld) can be seen searching the various embedded run-path search paths in order to find the required run-path-dependent dylib. Note that in this example scenario, the dylib is found in the second (i.e. non-primary) search directory (see Figure 18).

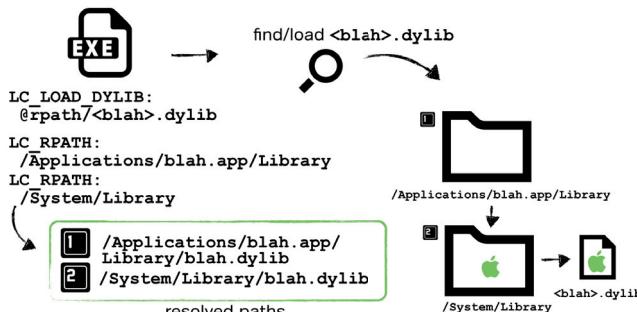


Figure 18: Dyld searching multiple run-path search directories.

The astute reader will recognize that this loader logic opens up yet another avenue for a dylib hijack attack. Specifically, if an application is linked against a run-path-dependent library, has multiple embedded run-path search paths, and the run-path-dependent library is not found in a primary search path, an attacker can perform a hijack. Such a hijack may be accomplished simply by ‘planting’ a malicious dylib into any of the primary run-path search paths. With the malicious dylib in place, any time the application is subsequently run, the loader will find the malicious dylib first, and load it blindly (see Figure 19).

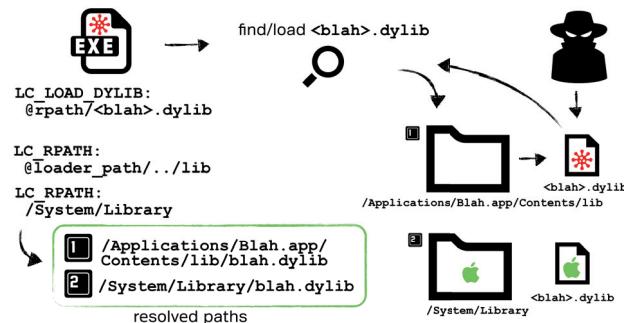


Figure 19: Hijacking an application via a malicious ‘@rpath’ dylib.

To summarize the findings so far: an OS X system is vulnerable to a hijacking attack given the presence of any application that:

1. Contains an LC\_LOAD\_WEAK\_DYLIB load command that references a non-existent dylib.
- or
2. Contains both an LC\_LOAD\*\_DYLIB load command that references a run-path-dependent library (@rpath) and multiple LC\_RPATH load commands, with the run-path-dependent library not found in a primary run-path search path.

The remainder of this paper will first walk through a complete dylib hijack attack, then present various attack scenarios (persistence, load-time process injection, ‘remote’ infection etc.), before concluding with some possible defences to counter such an attack.

In order to assist the reader in gaining a deeper understanding of dylib hijacking, it seems prudent to detail the trials, errors, and ultimate success of a hijack attack. Armed with this knowledge it will be trivial to understand attack automation, attack scenarios, and practical defences.

Recall the previously described sample application ('rPathApp.app') that was created in order to illustrate linking against a run-path-dependent dylib. This application will be the target of the hijack.

A dylib hijack is only possible against a vulnerable application (that is to say, one that fulfills either of the

two previously described hijack conditions). Since the example application (`rPathApp.app`) links against a run-path-dependent dylib, it may be vulnerable to the second hijack scenario. The simplest way to detect such a vulnerability is to enable debug logging in the loader, then simply run the application from the command line. To enable such logging, set the `DYLD_PRINT_RPATHS` environment variable. This will cause dyld to log its @rpath expansions and dylib loading attempts. Viewing this output should quickly reveal any vulnerable expansions (i.e. a primary expansion that points to a non-existent dylib), as shown in Figure 20.

```
$ export DYLD_PRINT_RPATHS="1"
$ ./Applications/rPathApp.app/Contents/MacOS/rPathApp
RPATH failed to expanding @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/One/
rpathLib.framework/Versions/A/rpathLib

RPATH successful expansion of @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/Two/
rpathLib.framework/Versions/A/rpathLib
```

Figure 20: The vulnerable (test) application, `rPathApp`.

Figure 20 shows the loader first looking for a required dylib (`rpathLib`) in a location where it does not exist. As was shown in Figure 19, in this scenario, an attacker could plant a malicious dylib in this primary run-path search path and the loader will then load it blindly.

A simple dylib was created to act as a malicious hijacker library. In order to gain automatic execution when loaded, the dylib implemented a constructor function. Such a constructor is executed automatically by the operating system when the dylib is loaded successfully. This is a nice feature to make use of, since generally code within a dylib isn't executed until the main application calls into it via some exported function.

```
_attribute__((constructor))
void customConstructor(int argc, const char **argv)
{
    //dbg msg
    syslog(LOG_ERR, "hijacker loaded in %s\n", argv[0]);
}
```

Figure 21: A dylib's constructor will automatically be executed.

Once compiled, this dylib was renamed to match the target (i.e. legitimate) library: `rpathlib`. Following this, the necessary directory structure (`Library/One/rpathLib.framework/Versions/A/`) was created and the ‘malicious’ dylib was copied in. This ensured that whenever the application was launched, dyld would now find (and load) the hijacker dylib during the search for the run-path dependent dylib.

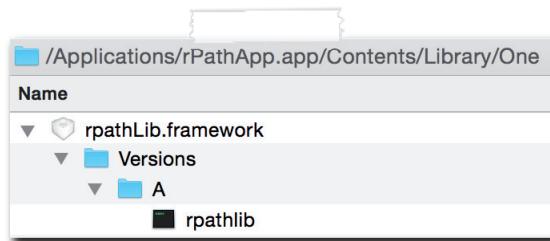


Figure 22: The ‘malicious’ dylib placed in the primary run-path search path.

Unfortunately, this initial hijack attempt failed and the application crashed miserably, as shown in Figure 23.

```
$ ./Applications/rPathApp.app/Contents/MacOS/rPathApp
RPATH successful expansion of @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/rPathApp

dYld: Library not loaded: @rpath/rpathLib.framework/Versions/A/rpathLib
Referenced from: /Applications/rPathApp.app/Contents/MacOS/rPathApp
Reason: Incompatible library version: rPathApp requires version 1.0.0
or later, but rpathLib provides version 0.0.0

Trace/BPT trap: 5
```

Figure 23: Success! Then crash and burning.

The good news, though, was that the loader found and attempted to load the hijacker dylib (see the ‘RPATH successful expansion...’ log message in Figure 23). And although the application crashed, this was preceded by an informative and verbose exception, thrown by dyld. The exception seemed self explanatory: the version of the hijacker dylib was not compatible with the required (or expected) version. Digging into the loader’s source code revealed the code that triggered this exception, as shown in Figure 24.

```
ImageLoader::recursiveLoadLibraries(...) {
    LibraryInfo actualInfo = dependentLib->doGetLibraryInfo();

    //compare version numbers
    if(actualInfo.minVersion < requiredLibInfo.info.minVersion)
    {
        //record values for use by CrashReporter or Finder
        dyld::throwf("Incompatible library version: .....");
    }
}

ImageLoaderMachO::doGetLibraryInfo() {
    LibraryInfo info;

    const dylib_command* dylibID = (dylib_command*)
        (&fMachOData[fDylibIDOffset]);

    //extract version info from LC_ID_DYLIB
    info.minVersion = dylibID->dylib.compatibility_version;
    info.maxVersion = dylibID->dylib.current_version;

    return info;
}
```

Figure 24: Dyld extracting and comparing compatibility version numbers.

As can be seen, the loader invokes the `doGetLibraryInfo()` method to extract compatibility and current version numbers from the `LC_ID_DYLIB` load command of the library that is being loaded. This extracted compatibility version number ('minVersion') is then checked against the version that the application requires. If it is too low, an incompatibility exception is thrown.

It was quite trivial to fix the compatibility issue (and thus prevent the exception) by updating the version numbers in Xcode, and then recompiling, as shown in Figure 25.



Figure 25: Setting the compatibility and current version numbers.

Dumping the `LC_ID_DYLIB` load command of the recompiled hijacker dylib confirmed the updated (and now compatible) version numbers, as shown in Figure 26.



Figure 26: Embedded compatibility and current version numbers.

The updated hijacker dylib was re-copied into the application's primary run-path search directory. Relaunching the vulnerable application again showed the loader 'finding' the hijacker dylib and attempting to load it. Alas, although the dylib was now seen as compatible (i.e. the version number checks passed), a new exception was thrown and the application crashed once again, as shown in Figure 27.

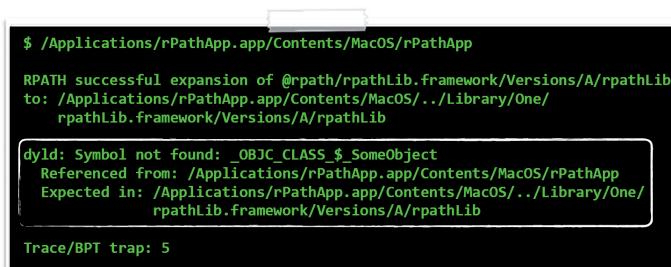


Figure 27: 'Symbol not found' exception.

Once again, the exception was quite verbose, explaining exactly why the loader threw it, and thus killed the application. Applications link against dependent libraries in order to access functionality (such as functions, objects, etc.) that are exported by the library. Once a required dylib is loaded into memory, the loader will attempt to resolve (via exported symbols) the required functionality that the dependent library is expected to export. If this functionality is not found, linking fails and the loading and linking process is aborted, thus crashing the process.

There were various ways to ensure that the hijacker dylib exported the correct symbols, such that it would be fully linked in. One naive approach would have been to implement and export code directly within the hijacker dylib to mimic all the exports of the target (legitimate) dylib. While this would probably have succeeded, it seemed complex and dylib specific (i.e. targeting another dylib would have required other exports). A more elegant approach was simply to instruct the linker to look elsewhere for the symbols it required. Of course, that elsewhere was the legitimate dylib. In this scenario, the hijacker dylib would simply act as a proxy or 're-exporter' dylib, and as the loader would follow its re-exporting directives, no linker errors would be thrown.

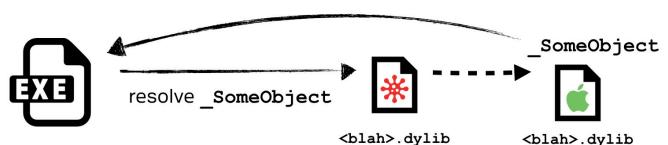


Figure 28: Re-exporting to the legitimate dylib.

It took some effort to get the re-exportation working seamlessly. The first step was to return to Xcode and add several linker flags to the hijacker dylib project. These flags included '`-Xlinker`', '`reexport_library`', and then the path to the target library which contained the actual exports that the vulnerable application was dependent upon.



Figure 29: Required linker flags to enable re-exporting.

These linker flags generated an embedded `LC_REEXPORT_DYLIB` load command that contained the path to the target (legitimate) library, as shown in Figure 30.

```
$ otool -l rPathLib
Load command 9
    cmd LC_REEXPORT_DYLIB
    cmdsize 72
        name @rpath/rpathLib.framework/Versions/A/rpathLib
```

Figure 30: Embedded LC\_REEXPORT\_DYLIB load command.

However, all was not well. Since the re-export target of the hijacker dylib was a run-path-dependent library, the name field in the embedded LC\_REEXPORT\_DYLIB (extracted from the legitimate dylib's LC\_ID\_DYLIB load command) began with '@rpath'. This was problematic since, unlike LC\_LOAD\*\_DYLIB load commands, dyld does not resolve run-path dependent paths in LC\_REEXPORT\_DYLIB load commands. In other words, the loader will try to load '@rpath/rpathLib.framework/Versions/A/rpathLib' directly from the file system. This, of course, would clearly fail.

The solution was to resolve the embedded '@rpath' path, providing the full path of the target library in the LC\_REEXPORT\_DYLIB load command. This was accomplished with one of Apple's developer tools: install\_name\_tool. To update the embedded install name (path) in the LC\_REEXPORT\_DYLIB load command, the tool was executed with the -change flag, the existing name (within the LC\_REEXPORT\_DYLIB), the new name, and finally the path to the hijacker dylib, as shown in Figure 31.

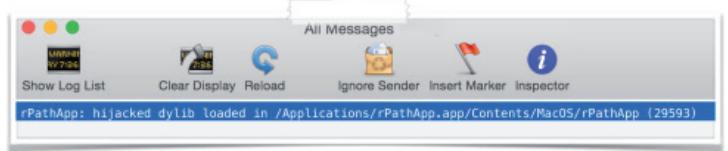
```
$ install_name_tool -change @rpath/rpathLib.framework/Versions/A/rpathLib /Applications/rPathApp.app/Contents/Library/Two/rpathLib.framework/Versions/A/rpathLib /Applications/rPathApp.app/Contents/Library/One/rpathLib.framework/Versions/A/rpathLib

$ otool -l Library/One/rpathLib.framework/Versions/A/rpathLib
Load command 9
    cmd LC_REEXPORT_DYLIB
    cmdsize 112
        name /Applications/rPathApp.app/Contents/Library/Two/rpathLib.framework/Versions/A/rpathLib
```

Figure 31: Using install\_tool\_name to update the embedded name (path).

With the path in the LC\_REEXPORT\_DYLIB load command updated correctly, the hijacked dylib was re-copied into the application's primary run-path search directory, and then the application was re-executed. As shown in Figure 32, this finally resulted in success.

To summarize: since the rPathApp application linked against a run-path-dependent library which was not found in the initial run-path search directory, it was vulnerable to a dylib hijack attack. Planting a specially compatible malicious dylib in the initial search path directory caused the loader to load the hijacker dylib blindly each time the application was



```
$ lsof -p 29593
COMMAND  NAME
rPathApp /Users/patrick
rPathApp /Applications/rPathApp.app/Contents/MacOS/rPathApp
rPathApp /Applications/rPathApp.app/Contents/Library/One/
    rpathLib.framework/Versions/A/rpathLib
rPathApp /Applications/rPathApp.app/Contents/Library/Two/
    rpathLib.framework/Versions/A/rpathLib
```

Figure 32: Successfully dylib hijacking a vulnerable application.

executed. Since the malicious dylib contained the correct versioning information as well as re-exporting all symbols to the legitimate dylib, all the required symbols were resolved, thus ensuring no functionality within the application was lost or broken.

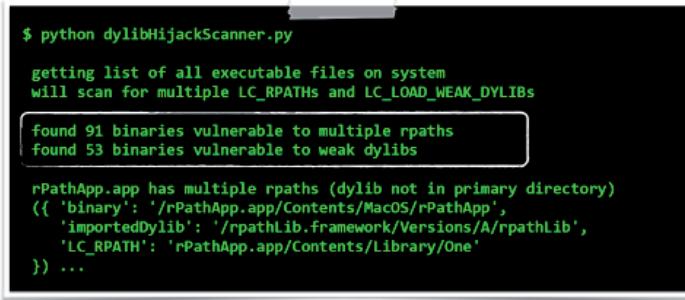
## ATTACKS

With a solid understanding of dylib hijacking on OS X behind us, it is now time to illustrate some real-life attack scenarios and provide some practical defences.

Advanced adversaries understand the importance of automating as many components of an attack as possible. Such automation increases scale and efficiency, freeing the attacker to focus on more demanding or complex aspects of the attack.

The first component of the hijack attack that was automated was the discovery of vulnerable applications. A Python script, dylibHijackScanner.py (available for download at [15]), was created to accomplish this task. After gathering either a list of running processes or all executables on the file system, the script intelligently parses the binaries' Mach-O headers and load commands. To detect binaries that may be hijacked via weak dylibs, the script looks for LC\_LOAD\_WEAK\_DYLIB load commands that reference non-existent dylibs. Automatically detecting binaries that may be hijacked due to non-existent @rpath'd imports was a little more complex. First, the script looks for a binary with at least one LC\_LOAD\*\_DYLIB load command that references a run-path-dependent dylib. If such a load command is found, the script continues parsing the binary's load commands looking for multiple LC\_RPATHs. In the case that both these prerequisites hold true, the script checks to see whether the run-path-dependent library import is found in a primary run-path search path. If the library does not exist, the script alerts the user that the binary is vulnerable. Executing the

scanner script revealed a surprising number of vulnerable applications, including (as expected) the vulnerable test application, rPathApp.app.



```
$ python dylibHijackScanner.py
getting list of all executable files on system
will scan for multiple LC_RPATHs and LC_LOAD_WEAK_DYLIBs
found 91 binaries vulnerable to multiple rpaths
found 53 binaries vulnerable to weak dylibs
rPathApp.app has multiple rpaths (dylib not in primary directory)
({ 'binary': '/rPathApp.app/Contents/MacOS/rPathApp',
  'importedDylib': '/rpathLib.framework/Versions/A/rpathLib',
  'LC_RPATH': 'rPathApp.app/Contents/Library/One'
}) ...
```

Figure 33: Automatically detecting vulnerable applications.

As can be seen in Figure 33, the scanner script found nearly 150 vulnerable binaries just on the author's work laptop! Interestingly, the majority of vulnerable applications fell into the more complex (from a prerequisite standpoint) 'multiple rpath' category. Due to space constraints, the full list of vulnerable applications cannot be shown here. However, Table 1 lists several of the more widespread or well-recognized applications that were found by the scanner script to be vulnerable to a dylib hijack.

Application	Company	Vulnerability
iCloud Photos	Apple	rpath import
Xcode	Apple	rpath import
Word	Microsoft	rpath & weak import
Excel	Microsoft	rpath & weak import
Google Drive	Google	rpath import
Java	Oracle	rpath import
PGP Keychain	PGP Tools	rpath import
Dropbox (garcon)	Dropbox	rpath import

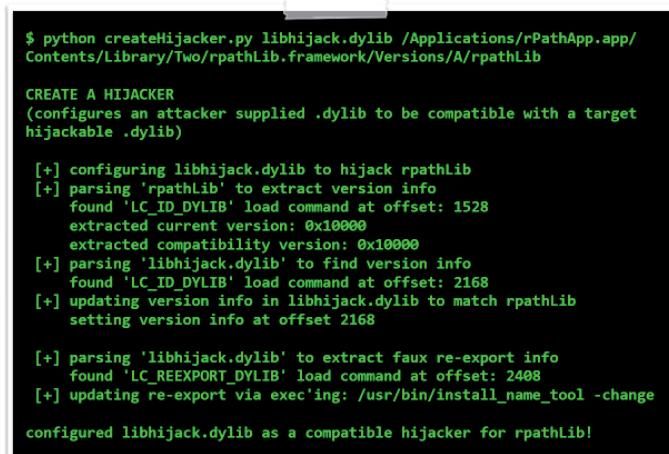
Table 1: Common vulnerable applications.

With an automated capability to uncover vulnerable applications, the next logical step was to automate the creation of compatible hijacker dylibs. Recall that two components of the hijacker dylib had to be customized in order to perform a hijack successfully. First, the hijacker dylib's versioning numbers had to be compatible with the legitimate dylib. Second (in the case of the rpath hijack), the hijacker dylib also had to contain a re-export (LC\_REEXPORT\_DYLIB) load command that pointed to the legitimate dylib, ensuring that all required symbols were resolvable.

It was fairly straightforward to automate the customization of a generic dylib to fulfil these two prerequisites. A second Python script, createHijacker.py (also available for download

at [15]), was created to perform this customization. First, the script finds and parses the relevant LC\_ID\_DYLIB load command within the target dylib (the legitimate dylib which the vulnerable application loads). This allows the necessary compatibility information to be extracted. Armed with this information, the hijacker dylib is similarly parsed, until its LC\_ID\_DYLIB load command is found. The script then updates the hijacker's LC\_ID\_DYLIB load command with the extracted compatibility information, thus ensuring a precise compatibility versioning match. Following this, the re-export issue is addressed by updating the hijacker dylib's LC\_REEXPORT\_DYLIB load command to point to the target dylib. While this could have been achieved by updating the LC\_REEXPORT\_DYLIB load command manually, it proved far easier simply to execute the `install_name_tool` command.

Figure 34 shows the Python script automatically configuring a generic hijacker dylib in order to exploit the vulnerable example application, rpathApp.app.



```
$ python createHijacker.py libhijack.dylib /Applications/rPathApp.app/
Contents/Library/Two/rpathLib.framework/Versions/A/rpathLib
CREATE A HIJACKER
(configures an attacker supplied .dylib to be compatible with a target
hijackable .dylib)

[+] configuring libhijack.dylib to hijack rpathLib
[+] parsing 'rpathLib' to extract version info
  found 'LC_ID_DYLIB' load command at offset: 1528
  extracted current version: 0x10000
  extracted compatibility version: 0x10000
[+] parsing 'libhijack.dylib' to find version info
  found 'LC_ID_DYLIB' load command at offset: 2168
[+] updating version info in libhijack.dylib to match rpathLib
  setting version info at offset 2168

[+] parsing 'libhijack.dylib' to extract faux re-export info
  found 'LC_REEXPORT_DYLIB' load command at offset: 2408
[+] updating re-export via exec'ing: /usr/bin/install_name_tool -change
configured libhijack.dylib as a compatible hijacker for rpathLib!
```

Figure 34: Automated hijacker creation.

Dylib hijacking can be used to perform a wide range of nefarious actions. This paper covers several of these, including persistence, load-time process injection, bypassing security products, and even a *Gatekeeper* bypass. These attacks, though highly damaging, are all realized simply by planting a malicious dylib which abuses legitimate functionality provided by the OS loader. As such, they are trivial to accomplish yet unlikely to be 'patched out' or even detected by personal security products.

Using dylib hijacking to achieve stealthy persistence is one of the most advantageous uses of the attack. If a vulnerable application is started automatically whenever the system is rebooted or the user logs in, a local attacker can perform a persistent dylib hijack to gain automatic execution of malicious code. Besides a novel persistence mechanism, this

scenario affords the attacker a fairly high level of stealth. First, it simply requires the planting of a single file – no OS components (e.g. startup configuration files or signed system binaries) are modified. This is important since such components are often monitored by security software or are trivial to verify. Second, the attacker’s dylib will be hosted within the context of an existing trusted process, making it difficult to detect as nothing will obviously appear amiss.

Of course, gaining such stealthy and elegant persistence requires a vulnerable application that is automatically started by the OS. Apple’s iCloud Photo Stream Agent (/Applications/iPhoto.app/Contents/Library/LoginItems/PhotoStreamAgent.app) is started automatically whenever a user logs in, in order to sync local content with the cloud. As luck would have it, the application contains multiple run-path search directories and several @rpath imports that are not found in the primary run-path search directory. In other words, it is vulnerable to a dylib hijack attack.

```
$ python dylibHijackScanner.py
PhotoStreamAgent is vulnerable (multiple rpaths)
'binary':      '/Applications/iPhoto.app/Contents/Library/LoginItems/
               PhotoStreamAgent.app/Contents/MacOS/PhotoStreamAgent'
'importedDylib': '/PhotoFoundation.framework/Versions/A/PhotoFoundation'
'LC_RPATH':     '/Applications/iPhoto.app/Contents/Library/LoginItems'
```

Figure 35: Apple’s vulnerable Photo Stream Agent.

Using the createHijacker.py script, it was trivial to configure a malicious hijacker dylib to ensure compatibility with the target dylib and application. It should be noted that in this case, since the vulnerable import (‘PhotoFoundation’) was found within a framework bundle, the same bundle structure was recreated in the primary run-path search directory (/Applications/iPhoto.app/Contents/Library/LoginItems/). With the correct bundle layout and malicious hijacker dylib (renamed as ‘PhotoFoundation’) placed within the primary run-path search directory, the loader found and loaded the malicious dylib whenever the iCloud Photo Stream Agent was started. Since this application was executed by the OS, the hijacker dylib was stealthily and surreptitiously persisted across reboots.

```
$ reboot
$ lsof -p <pid of PhotoStreamAgent>
/Applications/iPhoto.app/Contents/Library/LoginItems/
/PhotoFoundation.framework/Versions/A/PhotoFoundation
```

Figure 36: Hijacking Apple’s Photo Stream Agent for persistence.

As a final note on persistence, if no vulnerable applications are found to be started automatically by the OS, any vulnerable application commonly started by the user (such as a browser,

or mail client) may be targeted as well. Alternatively, a legitimate vulnerable application could easily be made persistent in a variety of ways (for example registering it as a Login Item, etc.), then persistently exploited. Although this latter scenario increases the visibility of the attack, the attacker dylib would, of course, prevent any UI from being displayed. Thus, it’s unlikely that the majority of users would notice a legitimate (Apple) binary automatically being started (and exploited) in the background.

Process injection, or coercing an external process into loading a dynamic library, is another useful attack scenario of dylib hijacking. In the context of this paper, ‘injection’ refers to load-time injection (i.e. whenever the process is started) as opposed to run-time injection. While the latter is arguably more powerful, the former is far simpler and often achieves the same level of damage.

Using dylib hijacking to coerce an external process into persistently loading a malicious dylib is a powerful and stealthy technique. As with the other dylib hijack attack scenarios, it does not require any modifications to OS components or binaries (e.g. patching the target process’s on-disk binary image). Moreover, since the planted dylib will persistently and automatically be loaded into the target process space each time the process is started, an attack no longer needs a separate monitoring component (to detect when the target process is started, then inject a malicious dylib). Also, since the attacker simply requires a malicious hijacker dylib to be planted, it neatly side-steps the complexities of run-time process injection. Finally, as this injection technique abuses legitimate functionality provided by the OS loader, it is unlikely to be detected by personal security products (which often attempt to prevent remote process injection by monitoring ‘inter-process’ APIs).

Xcode is Apple’s ‘Integrated Development Environment’ (IDE) application. It is used by developers to write both OS X and iOS applications. As such, it is a juicy target for an advanced adversary who may wish to inject code into its address space to surreptitiously infect the developer’s products (i.e. as a creative autonomous malware propagation mechanism). Xcode and several of its various helper tools and utilities are vulnerable to dylib hijack attacks. Specifically, run-path-dependent dylibs, such as DVTFoundation are not found in Xcode’s primary run-path search directories (see Figure 37).

The process injection hijack against Xcode was fairly straightforward to complete. First, a hijacker dylib was configured, such that its versioning information was compatible and it re-exported all symbols to the legitimate DVTFoundation. Then, the configured hijacker dylib was copied to /Applications/Xcode.app/Contents/Frameworks/DVTFoundation.framework/Versions/A/ (Frameworks/ being the primary run-path search directory). Now, whenever

```
$ python dylibHijackScanner.py
Xcode is vulnerable (multiple rpaths)
'binary':      '/Applications/Xcode.app/Contents/MacOS/Xcode'
'importedDylib': '/DVTFoundation.framework/Versions/A/DVTFoundation'
'LC_RPATH':     '/Applications/Xcode.app/Contents/Frameworks'
```

Figure 37: Apple's vulnerable IDE, Xcode.

Xcode was started, the malicious code was automatically loaded as well. Here, it was free to perform actions such as intercepting compile requests and surreptitiously injecting malicious source or binary code into the final products.

As Ken Thompson noted in his seminal work ‘Reflections on Trusting Trust’ [16], when you can’t trust the build process or compiler, you can’t even trust the code that you create.



Figure 38: Process ‘injection’ via dylib hijacking.

Besides persistence and load-time process injection, dylib hijacking can be used to bypass personal security products. Specifically, by leveraging a dylib hijack attack, an attacker can coerce a trusted process into automatically loading malicious code, then perform some previous blocked or ‘alertable’ action, now without detection.

Personal security products (PSPs) seek to detect malicious code via signatures, heuristic behavioural analysis, or simply by alerting the user whenever some event occurs. Since dylib hijacking is a novel technique that abuses legitimate functionality, both signature-based and heuristic-based products are trivial to bypass completely. However, security products, such as firewalls, that alert the user about any outgoing connections from an unknown process, pose more of a challenge to an attacker. Dylib hijacking can trivially thwart such products as well.

Personal firewalls are popular with *OS X* users. They often take a somewhat binary approach, fully trusting outgoing network connections from known processes, while alerting the user to any network activity originating from unknown or untrusted processes. While this is an effective method for detecting basic malware, advanced attackers can trivially bypass these products by exploiting their Achilles heel: trust. As mentioned, generally these products contain default rules, or allow the user to create blanket rules for known, trusted processes (e.g. ‘allow any outgoing connection from process X’). While this ensures that legitimate functionality is not broken, if an attacker can introduce malicious code

into the context of a trusted process, the code will inherit the process’s trust, and thus the firewall will allow its outgoing connections.

*GPG Tools* [17] is a message encryption suite for *OS X* that provides the ability to manage keys, send encrypted mail, or, via plug-ins, enable cryptographic services to arbitrary applications. Unfortunately, its products are susceptible to dylib hijacking.

```
$ python dylibHijackScanner.py
GPG Keychain is vulnerable (weak/rpath'd dylib)
'binary':      '/Applications/GPG Keychain.app/Contents/MacOS/GPG Keychain'
'weak dylib':   '/Libmacgpg.framework/Versions/B/Libmacgpg'
'LC_RPATH':    '/Applications/GPG Keychain.app/Contents/Frameworks'
```

Figure 39: GPG Tools’ vulnerable keychain app.

As *GPG Keychain* requires various Internet functionality (e.g. to look up keys on keyservers), it’s likely to have an ‘allow any outgoing connection’ rule, as shown in Figure 40.



Figure 40: Access rule for GPG Keychain.

Using a dylib hijack, an attacker can target the *GPG Keychain* application to load a malicious dylib into its address space. Here, the dylib will inherit the same level of trust as the process, and thus should be able to create outgoing connections without generating an alert. Testing this confirmed that the hijacker dylib was able to access the Internet in an uninhibited manner (see Figure 41).

Defensive-minded individuals may correctly point out that, in this scenario, *GPG Keychain*’s firewall rule could be tightened to mitigate this attack, by only allowing outgoing connections to specific remote endpoints (e.g. known key servers). However, there are a myriad of other vulnerable applications that may be hijacked to access the network in a similarly uninhibited manner. Or, in the case of the *Little Snitch* firewall, the inclusion of a system-level undetectable firewall rule allowing any connection from any process to talk to iCloud.com endpoints is more than enough for a full bypass (i.e. using a remote *iCloud iDrive* as a C&C server).

So far, the dylib attack scenarios described here have all been local. While they are powerful, elegant and stealthy, they all require existing access to a user’s computer. However, dylib hijacking can also be abused by a remote attacker in order to facilitate gaining initial access to a remote computer.

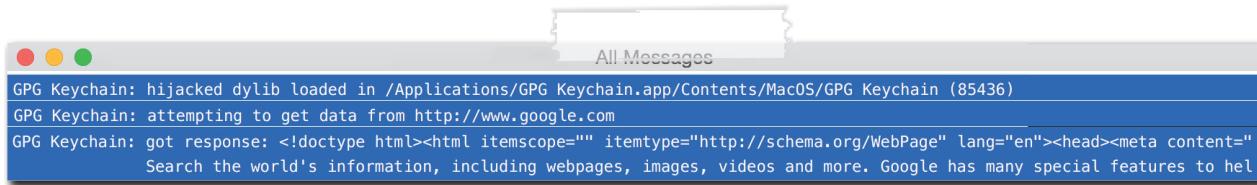


Figure 41: Bypassing a personal firewall (LittleSnitch) via dylib hijacking.

There are a variety of ways to infect Mac computers, but the simplest and most reliable is to deliver malicious content directly to end target(s). The ‘low-tech’ way is to coerce the user into downloading and installing the malicious content manually. Attackers creatively employ a range of techniques to accomplish this, such as providing ‘required’ plug-ins (to view content), fake updates or patches, fake security tools (‘rogue’ AV products), or even infected torrents.



Figure 42: Masked malicious content.

If the user is tricked into downloading and running any of this malicious content, they could become infected. While ‘low tech’, the success of such techniques should not be underestimated. In fact, when a rogue security program (Mac Defender) was distributed by such means, hundreds of thousands of OS X users were infected, with over 60,000 alone contacting AppleCare in order to resolve the issue [18].

Relying on trickery to infect a remote target will probably not work against more computer-savvy individuals. A more reliable (though far more advanced) technique relies on man-in-the-middling users’ connections as they download legitimate software. Due to the constraints of the *Mac App Store*, most software is still delivered via developer or company websites. If such software is downloaded via insecure connections (e.g. over HTTP), an attacker with the necessary level of network access may be able to infect the download in transit. When the user then runs the software, they will become infected, as shown in Figure 43.

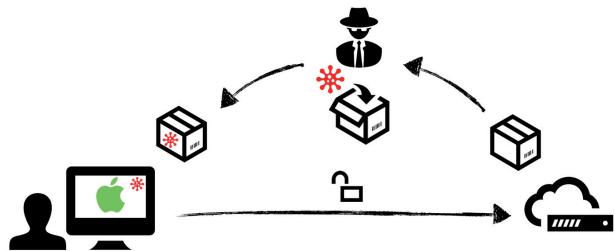


Figure 43: Man-in-the-middling a software download.

Readers may be thinking, ‘hey, it’s 2015, most software should be downloaded via secure channels, right?’ Unfortunately, even today, the majority of third-party OS X software is distributed insecurely. For example, of the software found installed in the author’s dock, 66% was distributed insecurely.



Figure 44: Software (in the author’s dock) that was distributed over HTTP.

Moreover, further research uncovered that all major third-party OS X security products were similarly distributed insecurely (see Figure 45).

*Apple* is well aware of these risks, and since version OS X *Lion* (10.7.5), Mac computers have shipped with a built-in security product, named *Gatekeeper*, that is designed to counter these attack vectors directly.

The concept of *Gatekeeper* is simple, yet highly effective: block any untrusted software from executing. Behind the scenes, things are a little more complex, but for the purposes of this discussion, a higher-level overview suffices. When any executable content is downloaded, it is tagged with a ‘quarantined’ attribute. The first time such content is set to run, *Gatekeeper* verifies the software. Depending on the user’s settings, if the software is not signed with a known *Apple* developer ID (default), or from the *Mac App Store*, *Gatekeeper* will disallow the application from executing.

With *Gatekeeper* automatically installed and enabled on all modern versions of OS X, tricking users into installing

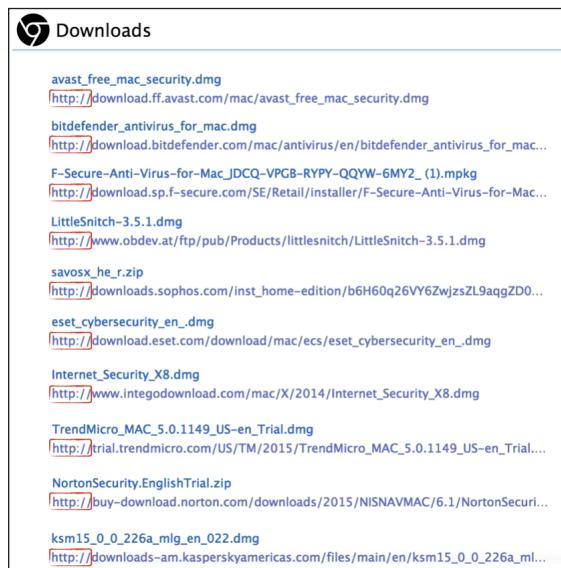


Figure 45: Insecure downloads of major OS X security products.

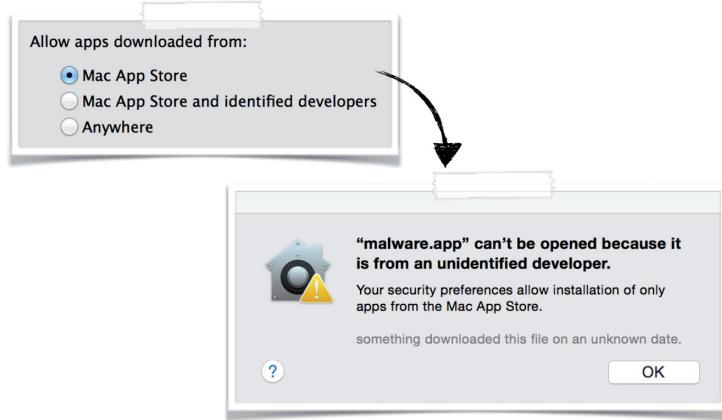


Figure 46: Gatekeeper in action.

malicious software or infecting insecure downloads (which will break digital signatures) is essentially fully mitigated. (Of course, an attacker could attempt to obtain a valid *Apple* developer certificate, then sign their malicious software. However, *Apple* is fairly cautious about handing out such certificates, and moreover, has an effective certificate revocation process that can block certificates if any abuse is discovered. Also, if *Gatekeeper* is set to only allow software from the *Mac App Store*, this abuse scenario is impossible.)

Unfortunately, by abusing a dylib hijack, an attacker can bypass *Gatekeeper* to run unsigned malicious code – even if the user's settings only allow *Apple*-signed code from the *Mac App Store*. This (re)opens the previously discussed attack vectors and puts *OS X* users at risk once again.

Conceptually, bypassing *Gatekeeper* via dylib hijacking is straightforward. While *Gatekeeper* fully validates the contents of software packages that are being executed (e.g. everything in an application bundle), it does not verify 'external' components.

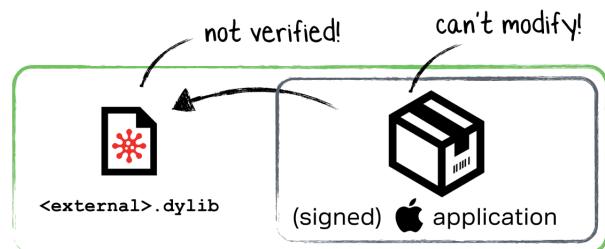


Figure 47: Theoretical dmg/zip that would bypass Gatekeeper.

Normally this isn't a problem – why would a downloaded (legitimate) application ever load relatively external content? (Hint: relative, yet external content.)

As *Gatekeeper* only verifies internal content, if an *Apple*-signed or *Mac App Store* application contains a relative external reference to a hijackable dylib, an attacker can bypass *Gatekeeper*. Specifically, the attacker can create (or infect in transit) a .dmg or .zip file with the necessary folder structure to contain the malicious dylib in the externally referenced relative location. When the legitimate application is executed by the unsuspecting user, *Gatekeeper* will verify the application bundle then (as it is trusted, and unmodified) allow it to execute. During the loading process, the dylib hijack will be triggered and the externally referenced malicious dylib will be loaded – even if *Gatekeeper* is set to only allow code from the *Mac App Store*!

Finding a vulnerable application that fulfils the necessary prerequisites was fairly easy. Instruments.app is an *Apple*-signed '*Gatekeeper approved*' application that expects to be installed within a sub-directory of Xcode.app. As such, it contains relative references to dylibs outside of its application bundle; dylibs that can be hijacked.

```
$ sudo spctl -vat execute /Applications/Xcode.app/Contents/Applications/Instruments.app
Instruments.app: accepted source=Apple System

$ otool -l Instruments.app/Contents/MacOS/Instruments
Load command 16
    cmd LC_LOAD_WEAK_DYLIB
    name @rpath/CoreSimulator.framework/Versions/A/CoreSimulator

Load command 30
    cmd LC_RPATH
    path @executable_path/../../../../SharedFrameworks
```

Figure 48: Apple's vulnerable Instruments app.

With a vulnerable trusted application, a malicious .dmg image was created that would trigger the *Gatekeeper* bypass.

First, the Instruments.app was placed into the image. Then an external directory structure was created that contained the malicious dylib (CoreSimulator.framework/Versions/A/CoreSimulator).

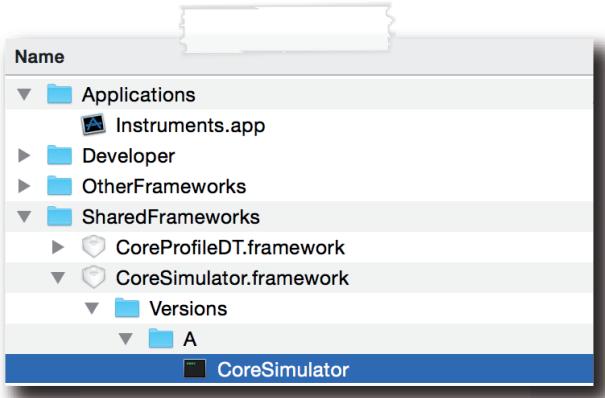


Figure 49: Malicious .dmg image.

To make the malicious .dmg more ‘believable’, the external files were set to hidden, a top level alias (with a custom icon) was created to point to Instruments.app, the background was changed, and the entire image was made read-only (so that it would automatically be displayed when double-clicked). The final product is shown in Figure 50.

This malicious (though seemingly benign) .dmg file was then ‘deployed’ (uploaded to a public URL) for testing purposes. When downloaded via *Safari* and then executed, *Gatekeeper*’s standard ‘this is downloaded from the Internet’ message window was initially shown. It is important to note that this alert is shown for any content downloaded from the Internet, and thus is not unusual.

Once this message window was dismissed, the malicious code was surreptitiously loaded along with the legitimate application. This, of course, should not have been allowed as *Gatekeeper*’s settings were at the maximum (only allow apps from the *Mac App Store*) (see Figure 51).



Figure 50: The finalized malicious .dmg image.

As the malicious dylib was loaded and executed before the application’s main method, the dylib could ensure that nothing appeared out of the ordinary. For example, in this case where the malicious .dmg masquerades as a Flash installer, the dylib can suppress Instruments.app’s UI, and instead spawn a legitimate Flash installer.

With the ability to bypass *Gatekeeper* and load unsigned malicious code, attackers can return to their old habits of tricking users into installing fake patches, updates or installers, fake AV products, or executing infected pirated applications. Worse yet, advanced adversaries with networking-level capabilities (who can intercept insecure connections) can now arbitrarily infect legitimate software downloads. Neither have to worry *Gatekeeper* any more.

## DEFENCES

Dylib hijacking is a powerful new attack class against OS X, that affords both local and remote attackers a wide range of malicious attack scenarios. Unfortunately, despite being contacted multiple times, Apple has shown no interest in addressing any of the issues described in this paper. Granted,

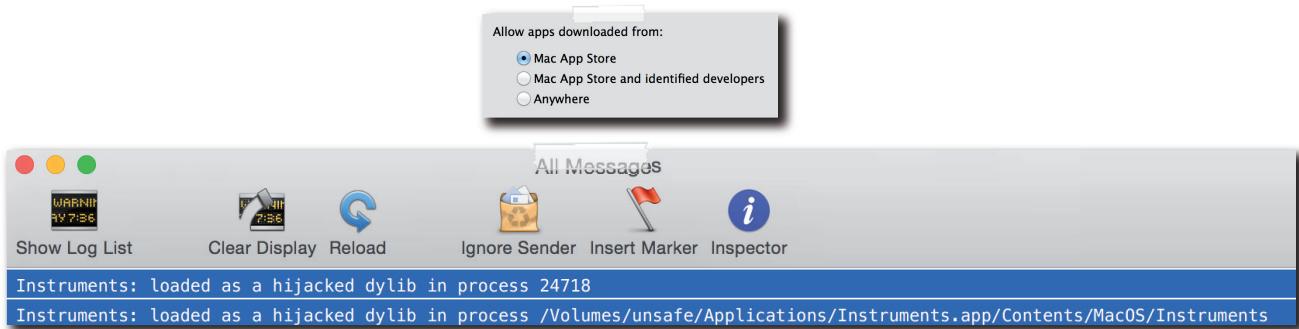


Figure 51: Bypassing Gatekeeper via a dylib hijack.

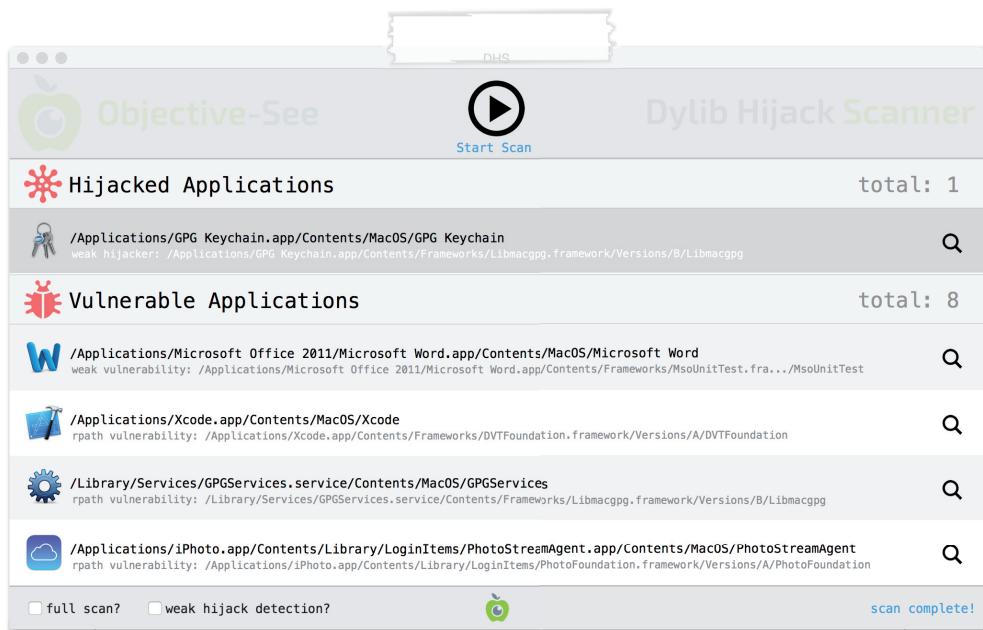


Figure 52: Objective-see's DHS scanner.

there appears to be no easy fix for the core issue of dylib hijacking as it abuses the legitimate functionality of the OS. However, it is the opinion of the author that *Gatekeeper* should certainly be fixed in order to prevent unsigned malicious code from executing.

Users may wonder what they can do to protect themselves. First, until *Gatekeeper* is fixed, downloading untrusted, or even legitimate software via insecure channels (e.g. via the Internet over HTTP) is not advised. Refraining from this will ensure that remote attackers will be unable to gain initial access to one's computer via the attack vector described in this paper. Due to the novelty of dylib hijacking on *OS X*, it is unlikely (though not impossible) that attackers or *OS X* malware are currently abusing such attacks locally. However, it can't hurt to be sure!

To detect local hijacks, as well as to reveal vulnerable applications, the author created a new application named *Dynamic Hijack Scanner* (or *DHS*). *DHS* attempts to uncover hijackers and vulnerable targets by scanning all running processes of the entire file-system. The application can be downloaded from objective-see.com.

## CONCLUSION

DLL hijacking is a well known attack class that affects the Windows OS. Until now, *OS X* was assumed to be immune to such attacks. This paper countered that assumption, illustrating a similar *OS X* attack, dubbed 'dylib hijacking'. By abusing weak or run-path-dependent imports, found

within countless *Apple* and third-party applications, this attack class opens up a multitude of attack scenarios to both local and remote attackers. From stealthy local persistence to a *Gatekeeper* bypass that provides avenues for remote infections, dylib hijacking is likely to become a powerful weapon in the arsenal of *OS X* attackers. And while *Apple* appears apathetic toward this novel attack, secure software downloads and tools such as *DHS* can ensure that *OS X* users remain secure... for now.

## REFERENCES

- [1] Secure loading of libraries to prevent DLL preloading attacks. [http://blogs.technet.com/cfs-file.ashx/\\_key/CommunityServer-Components-PostAttachments/00-03-35-14-21/Secure-loading-of-libraries-to-prevent-DLL-Preloading.docx](http://blogs.technet.com/cfs-file.ashx/_key/CommunityServer-Components-PostAttachments/00-03-35-14-21/Secure-loading-of-libraries-to-prevent-DLL-Preloading.docx).
- [2] DLL hijacking. [http://en.wikipedia.org/wiki/Dynamic-link\\_library#DLL\\_hijacking](http://en.wikipedia.org/wiki/Dynamic-link_library#DLL_hijacking).
- [3] Dynamic-Link Library Hijacking. <http://www.exploit-db.com/wp-content/themes/exploit/docs/31687.pdf>.
- [4] Windows NT Security Guidelines. <http://www.autistici.org/loa/pasky/NSAGuideV2.PDF>.
- [5] What the fxsst? <https://www.mandiant.com/blog/fxsst/>.
- [6] Leaked Carberp source code. <https://github.com/hzeroo/Carberp>.

- [7] Windows 7 UAC whitelist: Proof-of-concept source code. [http://www.pretentiousname.com/misc/W7E\\_Source/win7\\_uac\\_poc\\_details.html](http://www.pretentiousname.com/misc/W7E_Source/win7_uac_poc_details.html).
- [8] Microsoft Security Advisory 2269637; Insecure Library Loading Could Allow Remote Code Execution. <https://technet.microsoft.com/en-us/library/security/2269637.aspx>.
- [9] What is dll hijacking? <http://stackoverflow.com/a/3623571/3854841>.
- [10] OS X loader (dyld) source code. <http://www.opensource.apple.com/source/dyld>.
- [11] MachOView. <http://sourceforge.net/projects/machoview/>.
- [12] Run-Path Dependent Libraries. <https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/RunpathDependentLibraries.html>.
- [13] Using @rpath: Why and How. <http://www.dribin.org/dave/blog/archives/2009/11/15/rpath/>.
- [14] Friday Q&A 2012-11-09: dyld: Dynamic Linking On OS X. <https://www.mikeash.com/pyblog/friday-qa-2012-11-09-dyld-dynamic-linking-on-os-x.html>.
- [15] dylibHijackScanner.py & createHijacker.py. <https://github.com/synack/>.
- [16] Reflections on Trusting Trust. <http://cm.bell-labs.com/who/ken/trust.html>.
- [17] GPG Tools. <https://gpgtools.org/>.
- [18] Apple support to infected Mac users: ‘You cannot show the customer how to stop the process’. <https://nakedsecurity.sophos.com/2011/05/24/apple-support-to-infected-mac-users-you-cannot-show-the-customer-how-to-stop-the-process>.

---

**Editor:** Martijn Grooten  
**Chief of Operations:** John Hawes  
**Security Test Engineers:** Scott James, Tony Oliveira, Adrian Luca  
**Sales Executive:** Allison Sketchley  
**Editorial Assistant:** Helen Martin  
**Consultant Technical Editors:** Dr Morton Swimmer, Ian Whalley  
© 2015 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England.  
Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153  
Email: [editorial@virusbtn.com](mailto:editorial@virusbtn.com)  
Web: <http://www.virusbtn.com>

---