

Signing Prebound Executables on Mac OS X

Miroslav Jurišić, <meeroh@meeroh.org>

Abstract

Prebinding is an optimization present in Mac OS X which allows applications to launch faster. In the current implementation of prebinding, Mac OS X modifies executable files after they are installed. These modifications change cryptographic signatures of the executables, thus making it impractical to rely on cryptographic signatures to verify integrity of Mac OS X executables. This paper discusses how cryptographic signatures can be computed for Mac OS X executables in such a way that the signatures are not modified by prebinding.

Introduction

Prebinding is an optimization present in Mac OS X which allows applications to launch faster. The current implementation of prebinding requires executables installed on a Mac OS X system to be modified from time to time. This modification of executables causes their digests (checksums) and cryptographic signatures to change.

Changes to executables on a system are often used to detect that the system has been altered; for example, various intrusion detection systems monitor digests on system executables to detect an intruder trying to compromise the system. Therefore, the benign modifications introduced by prebinding cause false positives in intrusion detection systems.

In order to avoid this problem, we need to compute the digest of an application in a way which ignores the information affected by prebinding, without lowering the security of such a digest.

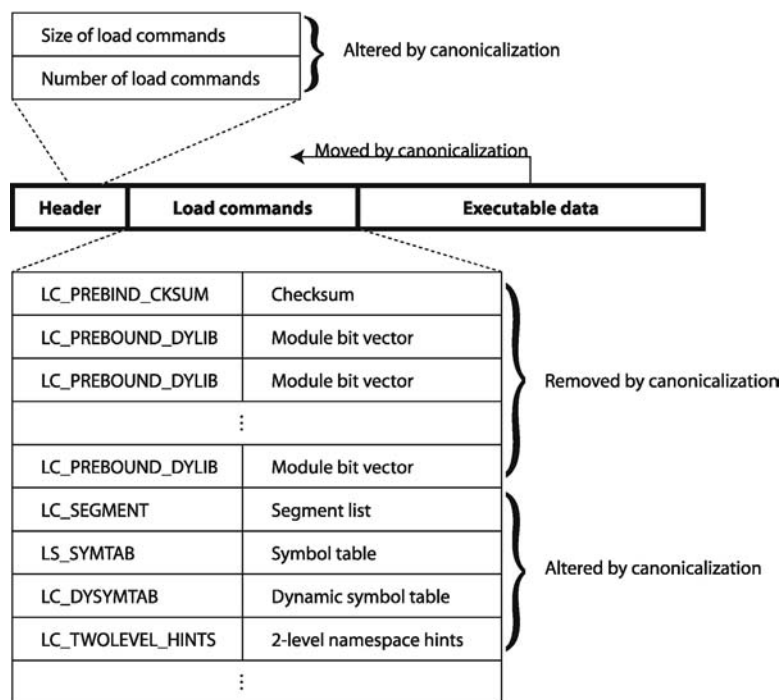
Analysis of Prebinding

Mach-O Executable Format

The Mach-O executable format is primarily documented in the <mach-o/loader.h> header file.

Each Mach-O executable consists of a header, followed by a number of load commands, followed by the the executable data. Each load command has a type, ranging from a simple LC_ID_DYLIB load command (which specifies the name and the version of a dynamic library), to a LC_SEGMENT load command (which specifies the location of various sections of the executable within the file).

Load commands can come in an arbitrary order in the file; not all executables have all possible load commands – indeed, some load commands only apply to certain types of executables (for example, an application would not have an LC_ID_DYLIB load command).



Canonicalization of the Mach-O executable format

Effects of Prebinding on Executables

Prebinding information is usually manipulated by `redo_prebinding` and `update_prebinding`. These are run explicitly by Installer (after an installation) or Software Update (after an update), or by `fix_prebinding` – a background process whose job is to monitor the system for executables which need prebinding.

In the process of updating prebinding for an executable, two kinds of load commands are modified: LC_PREBIND_CKSUM and LC_PREBOUND_DYLIB.

LC_PREBIND_CKSUM

The LC_PREBIND_CKSUM load command contains a digest of the executable. This digest is computed the first time prebinding is updated on the executable, and not modified thereafter. As a result, this digest is not affected by prebinding updates.

This digest attempts to address the same problem as this paper. For any purpose for which this digest is sufficiently reliable, it is the easiest way to obtain a stable digest of an executable.

However, the algorithm for this digest is not sufficient for all purposes. If we need to compute a digest of an executable using a specific algorithm, or do not consider the LC_PREBIND_CKSUM sufficiently strong, we need to be able to compute a new digest without relying on the one in LC_PREBIND_CKSUM.

The size of the LC_PREBIND_CKSUM digest is not changed by the prebinding process. There is always at most one LC_PREBIND_CKSUM in a given executable.

LC_PREBOUND_DYLIB

For each library an executable is linked against, the executable contains an LC_PREBOUND_DYLIB load command. This load command contains a bit vector, with one bit per module in the library. The bits specify whether a particular module is bound. Before the executable is first prebound, all the bits are usually zero; once the executable is prebound, some bits are set, to indicate that the executable is prebound to the corresponding module from the library.

The number of LC_PREBOUND_DYLIB commands can be changed by the prebinding process.

The size of each LC_PREBOUND_DYLIB command depends on the number of modules in the library it refers to, and therefore depends on the library executable itself. Hence, if a library executable changes, it is possible for the size or the contents of the LC_PREBOUND_DYLIB command for the library to change in any application linked against the library the next time the application's prebinding is updated.

Digests and Cryptographic Signatures

Typical digest and cryptographic signature algorithms operate on a byte stream by transforming it into a smaller piece of data which uniquely identifies the executable (to a very high probability). An example of such algorithm is a 32-bit CRC (such as the one implemented by the `sum 1` command) or MD5 (as implemented by the `md5sum 2` command).

Loosely, such algorithms have the property that small changes to the input produce large changes in the output, and that it is computationally infeasible to compute an input which would produce a given output. Those two properties together imply that it is computationally infeasible to produce a modified version of a file without altering its digest.

Canonical Executable Form

In order to have digests and signatures which do not change as an application is prebound (possibly several times), we need to define a canonical executable form, with the following properties:

- The canonical form of an executable does not change when the executable is prebound.
- Given two executables which differ in something other than their prebinding information, their canonical forms are different.
- The canonical form of an executable is itself a Mach-O executable.

These conditions guarantee that two executables are equivalent (i.e., do not differ other than possibly in their prebinding information) if and only if their canonical forms are equal, and that the canonical form of an executable is not equal to a file which is not a valid executable.

Now it remains to consider how prebinding information can be removed from an executable in a way that satisfies the above three conditions, thus giving us the canonical form of the executable.

LC_PREBIND_CKSUM

The `LC_PREBIND_CKSUM` command, if it exists, should be removed when generating the canonical form of the executable. This load command is only present in prebound executables, and in no way affects execution.

LC_PREBOUND_DYLIB

The number of `LC_PREBOUND_DYLIB` commands depends on the number of libraries that the executable has been prebound against. Since this includes not only the libraries against which the executable links directly, but also the libraries against which those libraries link (etc. ad nauseam), it is possible for a change in a dependent library to cause the number of `LC_PREBOUND_DYLIB` commands to change.

Therefore, we must not only ignore the contents of these commands (for they are bound to change then the executable is prebound) but also their number and size. Indeed, we need to completely ignore them when computing the canonical form of the executable.

Mach-O Executable Header

Given that two kinds of load commands are removed when computing the canonical executable form, we need to make adjustments to the Mach-O executable header, which contains the total number of load commands and their total length. The values in those fields need to be appropriately diminished.

If we were to leave the header unaltered, then any prebinding which resulted in a change to the number or length of `LC_PREBOUND_DYLIB` would cause the canonical form to change. (In addition to that, the canonical form would not be a valid Mach-O binary.)

Offsets to Executable Data

In the process of generating the canonical executable form, we removed some load commands. Since the length and number of those load commands can vary, we cannot simply replace those load commands with zeros to generate the executable form. We must actually remove the load commands.

Neither can we pad the segment of the executable containing load commands with zeros. It is possible for an executable to contain more than 4k of prebinding data, and an equivalent executable to contain less than 4k of prebinding data. Since all segments of an executable are aligned on 4k boundaries (to enable file-mapping), this means that the segments of those two executables might not start be at the same offsets. If we were to simply pad the segment containing the load commands, then these two equivalent executables' canonical forms would differ in length. However, the canonical form must have the same length for two equivalent executables.

It turns out that this is most easily accomplished if the canonical form is as short as possible. The reason for this is that lengthening any segment may require us to change the load address of any

segments that it may overlap once lengthened, which presents a big headache if the overlapped segment cannot easily be loaded at a different address, as is the case with a segment containing position-dependent code.

On the other hand, once we shorten a segment, the offset of segments which come after it change. Therefore, after we adjust the size of the segment containing the load commands (`__TEXT` segment), we must also adjust the offset of every segment that follows it.

Effect Of Canonicalization On Digests And Signatures

The ultimate purpose of the canonical form is to compute a digest or a signature, and therefore we must consider how the algorithm for generating the canonical form affects digests and signatures. An attacker seeking to subvert the digest or the signature may attempt to modify the executable in a way which can't be detected, or to generate an executable which has a particular digest or signature.

The attacker can add, remove, or alter any `LC_PREBIND_CKSUM` and `LC_PREBOUND_DYLIB` commands. In order to maintain the canonical form, the attacker also has to adjust the header and possibly the offsets.

This gives the attacker the ability to change disk and memory footprint of the executable, but not to affect the executable code itself; furthermore, the footprint can only be changed in increments of 4k.

Thus, the attacker can change the code path taken by prebinding and the dynamic loader, as well as the file mapping and VM paging subsystems of the operating system. If any of the parts of the operating system involved have bugs which depend on the contents of those two types of load commands, the attacker might be able to take advantage of those bugs by carefully crafting an executable which is equivalent to another executable, but with appropriate modifications to the data in the relevant load commands.

By changing the footprint of the application, the attacker can also execute a denial-of-service attack; an executable can be increased in size until it exceeds available memory on the system, and the user may no longer be able to launch it. Furthermore, since virtual memory is a shared resource, any other applications which do not behave predictably in face of virtual memory exhaustion may fail in a variety of ways, including crashing and losing data.

Implementation

The C++ code in Appendix implements a simple command-line tool `undo_prebinding` which accepts a Mach-O executable file on standard input and produces its canonical form on standard output. This output can be passed to any digest or digital signature algorithm, and it will produce the same result regardless of the prebinding state of the original executable.

This implementation does not assume any particular digest or signature mechanism; it works equally well with widely available ones (such as the ones available in OpenSSL), proprietary ones, or ones that haven't even been published yet.

Conclusion

By carefully filtering a Mach-O executable, it is possible to convert it to a canonical form and then generate a digest or a signature which does not change in presence of prebinding. This still leaves opportunity for some attacks, but they do not enable a potential attacker to easily execute arbitrary code without detection.

Even the remaining risks may be unacceptable in certain scenarios, and I therefore believe that prebinding should be redesigned to avoid modifying executables on a system. The information could be stored separately from the executable to improve security without compromising usability.

Revision History

- June 3, 2003: Final draft for MacHack 2003
- March 15, 2003: First draft

References

- [1] Prebinding
<<http://developer.apple.com/tools/projectbuilder/Prebinding.html>>

[2] man 1 sum

<<http://www.osxfaq.com/man/1/sum.ws>>

[3] man 1 md5sum

<<http://www.osxfaq.com/man/1/md5sum.ws>>

Appendix: Implementation

UndoPrebinding Namespace Documentation

UndoPrebinding Namespace Reference

Functions

- void UndoPrebinding (istream &inInput, ostream &inOutput)
Copy an executable from inInput to inOutput and remove prebinding information.

Function Documentation

[UndoPrebinding]void UndoPrebinding (istream & *inInput*, ostream & *inOutput*)

Copy an executable from inInput to inOutput and remove prebinding information.

- inInput input stream
- inOutput output stream

Definition at line 37 of file UndoPrebinding.cp.

```
40     {
41         inInput.exceptions (istream::eofbit
42             | istream::failbit | istream::badbit);
43         inOutput.exceptions (ostream::eofbit
44             | ostream::failbit | ostream::badbit);
45
46         // Get the mach header off the stream
47         mach_header    header;
48         inInput >> BinaryBlob (header);
49
50         // Check that this is a Mach-0 executable
51         if (header.magic != MH_MAGIC) {
52             Log::debug << "Header magic is "
53                 << setw (8)
54                 << setbase (16)
55                 << setfill ('0')
56                 << header.magic << endl;
57             throw runtime_error ("Not a valid Mach-0 file.");
58         }
59
60         if (header.filetype != MH_EXECUTE) {
61             Log::debug << "Header file type is "
62                 << header.filetype << endl;
63             throw runtime_error ("Not a Mach-0 executable.");
64         }
65
66         if ((header.flags & MH_PREBOUND) == 0) {
67             throw runtime_error ("Executable is not prebound.");
68         }
69
70         // Store data in the buffer until we have
71         // enough information to rewrite the header
72         stringstream    buffer;
73         buffer.exceptions (ostream::eofbit
74             | ostream::failbit | ostream::badbit);
75
76         unsigned long    currentSize = 0;
77         unsigned long    removedCommands = 0;
78         unsigned long    removedBytes = 0;
79
80         for (unsigned long cmd = 0; cmd < header.ncmds; ++cmd) {
81             load_command    command;
82             inInput >> BinaryBlob (command);
83
84             // Length of command data
85             unsigned long    dataSize = command.cmdsize;
```

```

86     dataSize -= sizeof (command);
87
88     switch (command.cmd) {
89         // Skip these two commands completely
90         case LC_PREBIND_CKSUM:
91             Log::debug << "Skipping LC_PREBIND_CKSUM "
92                 << " ("
93                 << command.cmdsize
94                 << " bytes)" << endl;
95             inInput.ignore (dataSize);
96             removedCommands++;
97             removedBytes += command.cmdsize;
98             break;
99
100        case LC_PREBOUND_DYLIB:
101            Log::debug << "Skipping LC_PREBOUND_DYLIB "
102                << " ("
103                << command.cmdsize
104                << " bytes)" << endl;
105            inInput.ignore (dataSize);
106            removedCommands++;
107            removedBytes += command.cmdsize;
108            break;
109
110        // Copy everything else to output
111        default:
112            Log::debug << "Copying load command "
113                << setw (8)
114                << setbase (16)
115                << setfill ('0')
116                << command.cmd
117                << " ("
118                << command.cmdsize
119                << " bytes)" << endl;
120            buffer << BinaryBlob (command);
121            CopyBytes (inInput, buffer, dataSize);
122        };
123
124        currentSize += command.cmdsize;
125    }
126
127    if (currentSize != header.sizeofcmds) {
128        Log::debug << "Header length "
129            << header.sizeofcmds
130            << " differs from file length "
131            << currentSize << endl;
132        throw runtime_error ("Header length mismatch");
133    }
134

```

```

135     Log::debug << "Removing "
136         << removedCommands
137         << " commands, "
138         << removedBytes
139         << " bytes" << endl;
140     header.ncmds -= removedCommands;
141     header.sizeofcmds -= removedBytes;
142     header.flags &= ~MH_PREBOUND;
143
144     buffer.seekg (0);
145
146     // Always remove data in page-sized blocks
147     unsigned long padBytes = removedBytes % 0x10000;
148     removedBytes -= padBytes;
149
150     Log::debug << "After rounding, "
151         << removedBytes
152         << " bytes will be removed, and "
153         << padBytes
154         << " bytes will be added as padding" << endl;
155
156     // Write the new header to output
157     inOutput << BinaryBlob (header);
158
159     // Copy the saved commands and adjust their offsets
160     for (unsigned long cmd = 0; cmd < header.ncmds; ++cmd) {
161         load_command      command;
162         buffer >> BinaryBlob (command);
163
164         // Length of command data
165         unsigned long      dataSize = command.cmdsize;
166         dataSize -= sizeof (command);
167
168         Log::debug << "Adjusting load command "
169             << setw (8)
170             << setbase (16)
171             << setfill ('0')
172             << command.cmd
173             << " at offset "
174             << sizeof (mach_header) + buffer.tellg ()
175             << endl;
176
177         switch (command.cmd) {
178             case LC_SEGMENT: {
179                 buffer.seekg (-sizeof (command), ios::cur);
180                 segment_command      segmentCommand;
181                 buffer >> BinaryBlob (segmentCommand);
182
183                 if (segmentCommand.fileoff > 0) {

```

```

184         // If the segment does not start at
185         // the beginning of the file, then
186         // its beginning will have shifted
187         segmentCommand.fileoff -= removedBytes;
188     } else if (segmentCommand.filesize > 0) {
189         // If the segment is at the beginning of
190         // the file and is not zero-length, then
191         // it includes load commands, and
192         // needs to be shortened
193         segmentCommand.filesize -= removedBytes;
194         segmentCommand.vmsize -= removedBytes;
195     }
196
197     inOutput << BinaryBlob (segmentCommand);
198
199     for (
200         unsigned long sectionIndex = 0;
201         sectionIndex < segmentCommand.nsects;
202         ++sectionIndex
203     ) {
204         section sectionData;
205         buffer >> BinaryBlob (sectionData);
206
207         if (sectionData.offset > 0) {
208             // If a section does not start at
209             // the beginning, it will have shifted
210             sectionData.offset -= removedBytes;
211         } else if (sectionData.size > 0) {
212             // If it starts at the beginning, and
213             // is not empty, it will have shrunk
214             sectionData.size -= removedBytes;
215         }
216
217         if (sectionData.reloff > 0) {
218             sectionData.reloff -= removedBytes;
219         }
220         inOutput << BinaryBlob (sectionData);
221     }
222 } break;
223
224 case LC_SYMTAB: {
225     buffer.seekg (-sizeof (command), ios::cur);
226     symtab_command symtabCommand;
227     buffer >> BinaryBlob (symtabCommand);
228
229     // Adjust the offsets
230     if (symtabCommand.symoff > 0) {
231         symtabCommand.symoff -= removedBytes;
232     }

```

```

233
234         if (symtabCommand.stroff > 0) {
235             symtabCommand.stroff -= removedBytes;
236         }
237         inOutput << BinaryBlob (symtabCommand);
238     } break;
239
240     case LC_DYSYMTAB: {
241         buffer.seekg (-sizeof (command), ios::cur);
242         dysymtab_command  dysymtabCommand;
243         buffer >> BinaryBlob (dysymtabCommand);
244
245         // Adjust the offsets
246         if (dysymtabCommand.tocoff > 0) {
247             dysymtabCommand.tocoff -= removedBytes;
248         }
249
250         if (dysymtabCommand.modtaboff > 0) {
251             dysymtabCommand.modtaboff -= removedBytes;
252         }
253
254         if (dysymtabCommand.extrefsymoff > 0) {
255             dysymtabCommand.extrefsymoff -= removedBytes;
256         }
257
258         if (dysymtabCommand.indirectsymoff > 0) {
259             dysymtabCommand.indirectsymoff -= removedBytes;
260         }
261
262         if (dysymtabCommand.extreloff > 0) {
263             dysymtabCommand.extreloff -= removedBytes;
264         }
265
266         if (dysymtabCommand.locreloff > 0) {
267             dysymtabCommand.locreloff -= removedBytes;
268         }
269
270         inOutput << BinaryBlob (dysymtabCommand);
271     } break;
272
273     case LC_PREBOUND_DYLIB: {
274         throw (logic_error ("LC_PREBOUND_DYLIB found"));
275     } break;
276
277     case LC_TWOLEVEL_HINTS: {
278         buffer.seekg (-sizeof (command), ios::cur);
279         twolevel_hints_command  twolevelHintsCommand;
280         buffer >> BinaryBlob (twolevelHintsCommand);
281

```

```

282         // Adjust the offsets
283         if (twolevelHintsCommand.offset > 0) {
284             twolevelHintsCommand.offset -= removedBytes;
285         }
286
287         inOutput << BinaryBlob (twolevelHintsCommand);
288     } break;
289
290     case LC_PREBIND_CKSUM: {
291         throw (logic_error ("LC_PREBIND_CKSUM found"));
292     } break;
293
294     // These load commands can be copied unchanged
295     case LC_UNIXTHREAD:
296     case LC_LOAD_DYLIB:
297     case LC_LOAD_DYLINKER: {
298         inOutput << BinaryBlob (command);
299         CopyBytes (buffer, inOutput, dataSize);
300     } break;
301
302     case LC_SYMSEG:
303     case LC_THREAD:
304     case LC_LOADFVMLIB:
305     case LC_IDFVMLIB:
306     case LC_IDENT:
307     case LC_FVMFILE:
308     case LC_PREPAGE:
309     case LC_ID_DYLIB:
310     case LC_ID_DYLINKER:
311     case LC_ROUTINES:
312     case LC_SUB_FRAMEWORK:
313     case LC_SUB_UMBRELLA:
314     case LC_SUB_CLIENT:
315     case LC_SUB_LIBRARY:
316     case LC_LOAD_WEAK_DYLIB: {
317         Log::debug << "An unimplemented load command "
318             << setw (8)
319             << setbase (16)
320             << setfill ('0')
321             << command.cmd
322             << " was encountered" << endl;
323         throw logic_error ("Unimplemented load command");
324     } break;
325
326     default:
327         Log::debug << "Unknown load command "
328             << setw (8)
329             << setbase (16)
330             << setfill ('0')

```

```
331         << command.cmd << endl;
332         throw logic_error ("Unknown load command");
333     };
334 }
335
336 for (int i = 0; i < padBytes; ++i) {
337     inOutput.write ("\0", 1);
338 }
339
340 // Copy the rest of the input to the output
341 inOutput << inInput.rdbuf ();
342 }
```