The **CryptSums** utility checksums the Mach segments of an application at compile time and inserts encrypted checksums into the binary so the application can verify at startup that it hasn't been damaged or tampered with.

Once properly installed, it allows you to simply do `'make secure'` to create a secure version of you application. You'll still be able to do standard `'make'` and `'make debug'` without interference from the checksum test. **CryptSums** provides a simple routine that your application can call at startup/runtime to check that the segments in memory still result in the same checksum as the file at compile time. You select which Mach segments and sections to verify.

Installation

You need the following files to use the **CryptSums** utility:

`Makefile.p*amble` -- These files can be copied or linked if you don't already have `Makefile.p*amble` files for your application. If you do, you'll need to edit the contents of these files into yours. If you're not using a InterfaceBuilder generated `Makefile`, you can use the *`example`* entry in the `Makefile` in this directory as an example of how to compile a secure application. The only change you have to make to these files is setting a new value for `CS_PASSWORD` at the top of `Makefile.preamble`.

`cryptsum.h` -- You'll need to include this file in your `*_main` file for an InterfaceBuilder generated application or in whatever file in which you call `cs-checkkey()`, usually the same file that defines `main()`. You should copy or link this file to one of your include directories or the same directory as your application source.

`cryptsums` -- This program generates the encryted checksum table for the application binary. It needs to be somewhere on your search path (eg. the directory where you're compiling or `/usr/local/bin`).

`dummy.cryptsums` -- This is a dummy encrypted checksum table that gets compiled into your application on the first pass so that the segment that holds the checksums will exist when the `Makefile` inserts the

actual table into the application binary via `segedit`. You should copy or link this file to the same directory as your application source.

`libcryptsum.a` -- This is the library the contains the `cs_checkkey()` function that your `main()` routine calls. You should copy or link this file to one of your library directories or the same directory as your application source.

`Example_main.m` -- In addition to the above files, you'll need to modify the `*_main.m` file that InterfaceBuilder generated to be similar to this file. Make sure to turn off the 'Generate main file' switch in the *Attributes* box of the *Project Inspector* panel of InterfaceBuilder that prevents it from overwritting the `*_main.m` file on subsequent saves. If you are not compiling an InterfaceBuilder-based application, you can use the example `main()` routine in the file `example.c` in this directory as a guide.

The sources for the **CryptSums** utility are also included so you can modify it to suit your needs.

Usage

Once you've installed the `Makefile.p*amble`, `cryptsum.h`, `cryptsums`, `dummy.cryptsums`, `libcryptsum.a` files and modified your `*_main.m` file, you should be able to generate a secure version of your application via `'make secure'`. It will leave the result in `$(NAME).secure` (similar to `$(NAME).debug` when you do `'make debug'`. You can run the `*.secure` application directly or rename it. You should also be able to `'make'` and `'make debug'` without interference from the checksum utility. Doing a `'make secure'` also creates a `$(NAME).unsecure` file, the `unsecure_obj` directory and the file `$(NAME).cryptsums`. These files will be removed by a `'make clean'`. Make sure not to delete the `dummy.cryptsums` file as it will be needed for subsequent remakes.

The `Makefile.preamble` file defines the symbol `SECURE` when you do `'make secure'`. You can use this with an `#ifdef` conditional in your code as is done in the `Example_main.m` and `example.c` example files.

The routine that your code calls at startup/runtime is:

```
int cs_checkkey(const char *segment, const char *section, const char *key)
```

Where *segment* is the name of the segment in which the section you want to check resides, eg. "__TEXT", *section* is the name of the section, eg. "__text" and *key* is the password that the checksums were encrypted with. With the files provides, this is set in the Makefile.preamble file and available to the program as the defined symbol PASSWORD since it is also needed by the Makefile.postamble file to generate the original checksum table.

The return values for cs_checkkey() are defined in the file cryptsum.h:

CS_SUCCESS = 0	The checksum is valid.
CS_NOMATCH	The encrypted checksums didn't match.
CS_NOSECTION	There is no such segment/section in memory.
CS_NOCHECKSUM	The checksum table segment is missing.
CS_NOENTRY	There is no checksum in the table for the given segment/section.
CERROR (-1, defined in <c.h>)	A Unix system call error occurred in cs_checkkey().

If a secure application is launched, and the checksums are correct, you can have it continue as usual. If the checksums are incorrect, you get back an error code and can act appropriately. The Example_main.m file, opens an NXAlertPanel on a checksum error and the process exits with an error.

How it works

When you do a 'make secure', the Makefile.postamble file first does (the equivalent of) a 'make unsecure' to generate a compiled version of your application with the SECURE flag defined and a dummy checksum table stored in a Mach segment. It then runs the cryptsums program on the unsecure binary to

generate the `$(NAME).cryptsum` file. This file has one segment/section pair per line along with the combination password/checksum encryption using standard Unix routines. This information is spliced into the binary using the `segedit` utility and the result is placed in the `$(NAME).secure` file.

At startup/runtime, when the `cs_checkkey()` routine is called, another checksum is done on the memory image of the segment/section being tested. This is encrypted with the same password and tested against the matching entry in the checksum table in the file binary. The result of that comparison is returned.

Notes

There are various segments/sections that you might want to checksum, eg. `__TEXT/__text` to see if the compile code has been damaged/changed, `__TEXT/__cstring` to check the strings, etc. You can also check `__NIB/$(NAME).nib` to verify the InterfaceBuilder data though this is usually a reasonable section for users to modify to suit their needs/tastes. The `cryptsums` program generates encrypted checksums for all the sections at compile time but not all can be successfully verified at startup/runtime since some are legitimately changed at load/run time. Call `cs_checkkey()` on static sections that are vital to proper program behavior.

Make sure to set the `CS_PASSWORD` variable in `Makefile.preamble` to your own value. For security purpose, you should set it to a random string or something similar to what the Unix `strings` command generates when run on your application, eg. `'lockFocus'`, so that the password will difficult to determine via examining the binary.

The checksum routine defined in the `cryptsum.c` file is simple, though the encryption is complex. If you want a more sophisticated checksum, you can modify the routine `cs_checksum()` in the `cryptsum.c` file:

```
unsigned short cs_checksum(unsigned short checksum, unsigned char byte)
```

This function takes the current checksum, and a new byte to factor in and returns the updated checksum.

Since `cs_checkkey()` has to read an entire section to checksum it, you'll be loading that section into virtual memory, possibly unnecessarily for execution purposes, possibly into a different space (data vs. code). If your application is very, very large this could be of concern so you might want to only do `cs_checkkey()` when the application runs the first time and *installs* itself, and bypass the call for subsequent startups. For programs that already do initializations on the initial execution, adding the checksum call as an additional step should be trivial.

If your application has more than ~200 separate sections (InterfaceBuilder has 172), you may overflow the checksum table. To get around this, pad the `dummy.cryptsums` file (eg. duplicate the first line multiple times) until it exceeds 8192 bytes in length and change the constant `MAXBSIZE` in `cs_checkkey()` to be `2 * MAXBSIZE`. That should approximately double the number of sections you can checksum.

CryptSums isn't compatible with the `-object` flag to `ld` which suppresses Mach segments.

References

Mach-O (5), segedit (1), size (1), getsectdata(3), crypt(3) UNIX Programmer's Manual

`/NextLibrary/Documentation/NextDev/DevTools/08_MachO`