Issues with FLIRT aware malware

John McMaster

January 12, 2011

Abstract

Hex-Ray's Interactive Disassembler Pro (IDA) relies on the Fast Library Recognition Technology (FLIRT) algorithm to perform function recognition. However, FLIRT has become dated and is vulnerable to attack by malicious software (malware), copy protected software, or other reverse engineering hardened software. By attacking various parts of the signature, it is possible to make custom malicious code appear as if it is stock library code. As certain C runtime (CRT) environment functions are statically compiled into programs and have their signatures automatically loaded as well, it should be possible to turn this on the analyst to hinder analysis.

1 Background

Analyzing malicious software and other binaries is very time consuming. In order to speed up manual analysis, analysts rely on heuristic based automated analysis. In particular, it is very beneficial to recognize stock library functions since their purpose is already known. One such algorithm to achieve this goal is to use the FLIRT algorithm. While signature collisions are undesirable during creation and best avoided, the focus of the paper will be on how to intentionally create collisions against FLIRT signatures. it will be shown FLIRT is weak against collisions and can be turned against the analyst. Instead of saving the analyst time by eliminating monotonous work, a bad signature can create a false sense of security and severely hinder analysis.

2 Previous work

Several groups have worked with alternative function recognition algorithms. This may indicate these groups do not feel FLIRT is adequate for many applications, especially in malware where this research is typically seen. No public statements have been noted about the weaknesses in FLIRT. However, I did receive in private communications an acknowledgment that FLIRT may be weak.

3 FLIRT

Hex-Rays' Interactive Disassembler (IDA) Pro is a popular binary analysis program self-described as a ...hosted multi-processor disassembler and debugger that offers so many features it is hard to describe them all. [4] FLIRT is a function recognition algorithm designed to be used with IDA with the goal [t]o assist IDA users we (Hex-Rays) attempted to create an algorithm to recognize the standard library functions. [3] FLIRT function signatures are made through the FLAIR toolkit. [2] One of their goals is we try to avoid false positives completely. [3] FLIRT signatures are distributed in a proprietary binary .sig format. However, an open source equivalent implementation of FLAIR exists in uvudec that allows the format to be studied. [5]

To make a .sig file, an object file is run through an object to .pat conversion program such as pelf in FLAIR or uvobj2pat in uvudec. The .pat file is human readable and creates a signature for every module found in the input file. A module is the smallest collection of functions that are to be found together. For example, if a .a file is given, a module is the individual object files within the .a file. Example .pat line:

```
D9EDD9442404D9E5DFE0D9C09E7219D9
E1D81D........DFE09E7209D805....
00 0000 002F :0000 ___log1pf :0000@
_log1pf ....D9F1C3D9F9C37AE5DDD9DDD9C3
```

Each one of these modules has the following attributes in given order:

- leading thirty-two bytes represented in uppercase hexadecimal
- relocation bytes represented as ".."
- number of bytes included in the CRC16
- CRC16 of up to two hundred fifty-six bytes following the first thirty-two that are relocation free

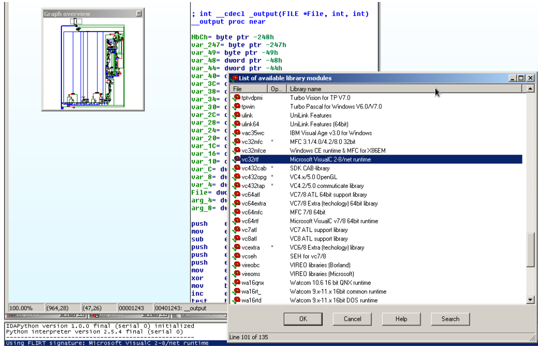

Figure 1: Visual Studio C Runtime environment (CRT) signatures being automatically loaded

- total length of the module
- public names in the module, prefixed with ":", "@" indicates a local symbol
- offset and name of the first relocation for each symbol prefixed with ˆ
- Tailing bytes that were not included in either the leading thirty-two or the CRC16

Next, the .pat file is translated into a .sig file.[1] The following occurs during conversion:

- All tailing bytes are discarded
- Only the first relocation name is recorded
- If two signatures are identical, the collision must be resolved, often by not including the signature

Finally, we note that IDA automatically loads certain .sig files. The .sig directory contains a file called vc32rtf.sig that is automatically loaded by IDA, as seen in the log window. Other .sig files can be loaded as well, but a practical attack should probably focus on those automatically loaded.

4 CCITT CRC16

Regardless of cryptographic strength, a 16 bit hash is extremely small by today's cryptographic standards notwithstanding that [b]oth CRCs and cryptographic hash functions by themselves do not protect against intentional modification of data. [1] Even if CRC16 was cryptographically secure, its combined short length and ease of computation makes it an easy target for brute force collision attacks.

[1]The exact details of the .sig format will not be covered as they are irrelevant to this discussion and can be found in the references

5 Flow analysis and other considerations

Since FLIRT only recognizes plausibly compiler generated functions, careful editing is required to prevent issues in the problems window. First, multiple disassembly paths raises a problem, so jumps must not interleave instructions. For example, if a jump from the prefix bytes intersects the payload, align it by inserting nop's, rearranging instructions, etc. Second, one must be careful not to truncate instructions, especially from a hard cutoff after the first thirty-two leading bytes. In the crc16 function example, the last byte is part of a stack pointer manipulation (1e: 8b 45 08 mov 0x8(%ebp),%eax). Substituting 0x08 with 0x90 makes the stack pointer analysis incoherent and IDA will refuse to create a function. This also indicates stack pointer analysis must remain valid, including function cleanup by restoring the stack and performing an actual return. Finally, IDA has some knowledge of Linux system calls. An initial collision attack attempted to use exit to avoid stack cleanup, but IDA detected the no-return style function call and resulted in incoherent function analysis.

Although subsequent relocations are ignored, they still must be cared for, especially if we are attacking the CRC16 block. In particular, a relative function call seems to still count as a relocatable even though the operand is fixed. Thus, relative function calls will truncate the CRC16 despite being constant data.

Finally, it should be noted that in general the entire library will be available as an attack surface, not a single function. Thus, in all likelihood, there will exist a number of possibilities to inject code and a single attack vector is not critical. As an example, vc32rtf.sig contains tens of thousands of signatures, leading to a very large attack surface. It is unknown, however, the effect of using two distinct signatures that resolve to the same function name and only unique function names may be viable. The attack surface is still very large and should not be an issue.

6 Forging signatures

6.1 Forging by CRC16

Consider the crc16 function sampled from uvudec.[2] It is a relatively long function that makes no external function calls as would get grouped into the crc16 block in a FLIRT signature. Thus it is a good

[2]See appendix for source code and disassembly dump of both the original crc16 function and the forgery.

example candidate for this style attack. The original function was compiled with gcc and ran through uvobj2pat:

```
5589E583EC1066C745FEFFFF837D0C00
75080FB745FEF7D0EB7FC645FB008B45
7B BF3F 009B :0000 _crc16
```

In order to create a function that looks identical, but performs differently, we have these fundamental constraints: -Must match the CRC16 -Must match the same overall function length -Must have the same relocations (none in this case) We gain the flexibility to forge the CRC16 with a statement like the following:

```
080484DF: jmp short loc_80484E3
080484E1: db 27h, 78h
080484E3: ...
```

This jump allocates 16 bits of freedom for a CRC16 collision without effecting program execution. This is better than, for example, creating a conditional branch on those bytes since IDA will disassemble the collision bytes and possibly result in invalid disassembly. It may be possible to forge the CRC16 with fewer bits, but this guarantees us a collision exists.

The relocation limitation may at first sound daunting since function calls cannot be made. However, one easy way around this is to make syscalls directly as they do not depend on loaded code locations. Another way might be to use retc style calls, although IDA might not like the flow control and this would be much more difficult.

Linking one program with the actual crc16 function (normal) and one with the collision (forged), both execute with different results:

```
[mcmaster@gespenst research]$ ./normal
crc16: 0x1F09
[mcmaster@gespenst research]$ ./forged
hello
crc16: 0x0006
```

Next, it will be shown IDA does not differentiate between the two programs or, more precisely, the crc16 function signatures. As library recognition is irrelevant for non-stripped executables, they are stripped of symbols. Then normal is loaded into IDA and the FLIRT signature for crc16 is loaded. The crc16 function is correctly discovered and indicated to be a library function as shown in light blue at the top of the window in figure 2.

Now, loading the forged executable in, a crc16 function is also recognized, but this time mistakenly: Thus, the attack has succeeded and the function is now mistakenly labeled.

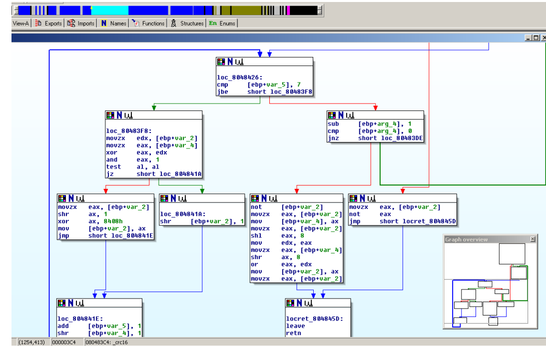

Figure 2: Good signature match

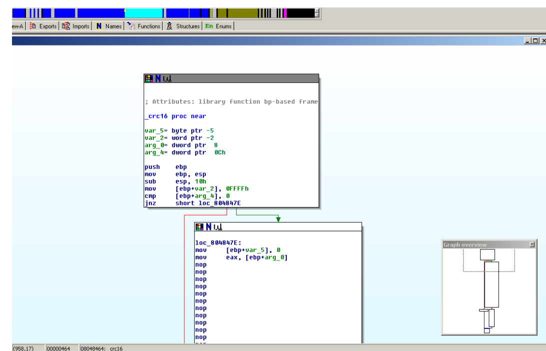

Figure 3: Forged signature match

6.2 Forging by function length

Since the remainder of the function is only based on length, it is trivial to forge the remainder as long as the assembly is reasonable looking as described above. The only main restriction is that the forged function have the same number of bytes as the original. If there is a function call to be ignored, it is not permissible to unconditionally jump over the call since it would then not be considered a relocation and the signature would fail. However, it is not difficult to simulate an unconditional jump in such a way that appears conditional to simple static analysis:

```
xor %eax, %eax
cmp %eax
je post_call
call some_function
post_call:
# rest of program ...
```

7 Impact

With function recogniztion no longer accurate, functions can no longer to be trusted as recognized. Soft-

ware expected to be FLIRT aware should take due note that function hints may be misleading.

It might also be possible to design a program to transform a program into code that IDA completly recognizes as library functions. However, if the purpose was to confuse the analyst by not knowing which are library functions and which aren't, it is probably easier to obfuscate the code by other methods. However, this may have interesting effects on IDA's ability to process a file.

8 Countermeasures

Several others have attempted to do more advanced function recognition schemes such as the works of Silvio Cesare and Yang Xiang (A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost). Most of these are based around control flows graphs within the function. FLIRT does this to a very limited aspect with its ability to record a single external function call, but was shown to be insufficient. Using flow graph approaches, the crc16 function may have been much more difficult to forge since all of the blocks are very small. However, some samples, such as our example payload do not require branches. By taking advantage of limitations of static analysis, it should be possible to view conditional branches as opportunities instead of absolutes. This may allow allow the flexibility necessary to forge a flowgraph based signature.

It should be noted, however, recording more information about a function will come at the cost of recognizing fewer variations of that function. Since it is generally better to error on the side of caution in function recognition (FLIRT), this may be an acceptable compromise.

9 Future work

Several improvements could be made to the attack. First, proper stack cleanup could be done with minimal effort. Additionally, since this program only calls crc16 with a fixed value, this forged value could be returned to maintain functionality. A way to accomplish all of this more easily may be to do a return to libc style call where the stack is modified in the forged library function to make a function call upon return. This technique would be useful since it could likely be setup in such a way that static analysis would not reveal the function call. Since IDA does do a somewhat detailed stack analysis, something like dynamic pointer math may be required to confuse it.

Additionally, the simple nop sled is very obvious to an analyst. If nothing else, a simple entropy analysis will show the forged area differs significantly from the rest of the program. This can be solved, for example, by introducing meaningless statements to marshal data around registers or jump conditionally to areas that also do nothing. This would make at least a quick glaze over the instruction sequence to seem legitimate library code.

10 Conclusions

This paper has given a background on FLIRT and shown why it is weak to collisions. Collisions can be generated by using the CRC16 section or by taking advantage of the lack of checks performed on the tailing bytes. The exact match nature of a thirty-two byte or less function makes them resistant to attacks. FLIRT attacks might be mitigated by recording more flow information within the function at a tradeoff of not recognizing some variations of that function. Ultimately, any signature based technique can likely be attacked and its only a question of how resistant they are. FLIRT is showing its age by using a cryptographically insecure hashing algorithm and topping it off with a potentially large wild-card block. While FLIRT will remain useful for reverse engineering code generated with commercial off the shelf compilers, software employing anti-reverse engineering mechanisms such as viruses and copy protection schemes may employ these techniques to hinder analysis.

References

[1] Ccso.com. Cyclic redundancy check, wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Cyclic_redundancy_check`.

[2] Chris Eagle. *The IDA Pro Book: the Unofficial Guide to the World's Most Popular Disassembler*. No Starch, 2008.

[3] Ilfak Guilfanov. Fast library identification and recognition technology. `http://www.hex-rays.com/idapro/flirt.htm`.

[4] Hex-Rays. Ida pro disassembler - multi-processor, windows hosted disassembler and debugger. `http://www.hex-rays.com/idapro/`.

[5] John McMaster. uvudec wiki. `https://github.com/JohnDMcMaster/uvudec/wiki`.

Appendices

A Original crc16 function

The two most popular CRC16 implementations seem to be this and a table based method. This implementation has been seen across the Internet in many locations. As such, I do not really know who wrote it and the site that I got it from no longer hosts it.

```
uint16_t crc16(const char *d_p,
    uint32_t length) {
  uint8_t i;
  uint16_t d, crc = 0xffff;
  if (length == 0) return (~crc);
  do {
    for (i=0, d=0xff & *d_p++;
         i < 8; ++i, d >>= 1) {
      if ((crc & 0x0001) ^ (d & 0x0001))
        crc = (crc >> 1) ^ 0x8408;
      else  crc >>= 1;
    }
  } while (--length);
  crc = ~crc;
  d = crc;
  crc = (crc << 8) | (d >> 8 & 0xff);
  return (crc);
}
```

Disassembly:

```
00000000 <_crc16>:
0: 55 push %ebp
1: 89 e5 mov %esp,%ebp
3: 83 ec 10 sub $0x10,%esp
6: 66 c7 45 fe ff ff
  movw $0xffff,-0x2(%ebp)
c: 83 7d 0c 00 cmpl $0x0,0xc(%ebp)
10: 75 08 jne 1a <crc16+0x1a>
12: 0f b7 45 fe movzwl -0x2(%ebp),%eax
16: f7 d0 not %eax
18: eb 7f jmp 99 <crc16+0x99>
1a: c6 45 fb 00 movb $0x0,-0x5(%ebp)
1e: 8b 45 08 mov 0x8(%ebp),%eax
# End leading bytes
21: 0f b6 00 movzbl (%eax),%eax
24: 66 98 cbtw
26: 66 25 ff 00 and $0xff,%ax
2a: 66 89 45 fc mov %ax,-0x4(%ebp)
2e: 83 45 08 01 addl $0x1,0x8(%ebp)
32: eb 2e jmp 62 <crc16+0x62>
34: 0f b7 55 fe movzwl -0x2(%ebp),%edx
38: 0f b7 45 fc movzwl -0x4(%ebp),%eax
3c: 31 d0 xor %edx,%eax
3e: 83 e0 01 and $0x1,%eax
41: 84 c0 test %al,%al
43: 74 11 je 56 <crc16+0x56>
45: 0f b7 45 fe movzwl -0x2(%ebp),%eax
49: 66 d1 e8 shr %ax
4c: 66 35 08 84 xor $0x8408,%ax
50: 66 89 45 fe mov %ax,-0x2(%ebp)
54: eb 04 jmp 5a <crc16+0x5a>
56: 66 d1 6d fe shrw -0x2(%ebp)
5a: 80 45 fb 01 addb $0x1,-0x5(%ebp)
5e: 66 d1 6d fc shrw -0x4(%ebp)
62: 80 7d fb 07 cmpb $0x7,-0x5(%ebp)
66: 76 cc jbe 34 <crc16+0x34>
68: 83 6d 0c 01 subl $0x1,0xc(%ebp)
6c: 83 7d 0c 00 cmpl $0x0,0xc(%ebp)
70: 75 a8 jne 1a <crc16+0x1a>
72: 66 f7 55 fe notw -0x2(%ebp)
76: 0f b7 45 fe movzwl -0x2(%ebp),%eax
7a: 66 89 45 fc mov %ax,-0x4(%ebp)
7e: 0f b7 45 fe movzwl -0x2(%ebp),%eax
82: c1 e0 08 shl $0x8,%eax
85: 89 c2 mov %eax,%edx
87: 0f b7 45 fc movzwl -0x4(%ebp),%eax
8b: 66 c1 e8 08 shr $0x8,%ax
8f: 09 d0 or %edx,%eax
91: 66 89 45 fe mov %ax,-0x2(%ebp)
95: 0f b7 45 fe movzwl -0x2(%ebp),%eax
99: c9 leave
9a: c3 ret
```

B Test program

```
#include <stdio.h>
#include "crc.h"
int main(void) {
  char to_hash[] = "Already tried "
    "a SIGQUIT, so now it's "
    "KILL DASH 9.";
  printf("crc16: 0x%04X\n",
    crc16(&to_hash[0],
      sizeof(to_hash)));
  return 0;
}
```

C A crc16 collision

```
00000000 <crc16>:
0: 55 push %ebp
1: 89 e5 mov %esp,%ebp
3: 83 ec 10 sub $0x10,%esp
6: 66 c7 45 fe ff ff movw $0xffff,-0x2(%ebp)
c: 83 7d 0c 00 cmpl $0x0,0xc(%ebp)
10: 75 08 jne 1a <crc16+0x1a>
```

```
12: 0f b7 45 fe movzwl -0x2(%ebp),%eax
16: f7 d0 not %eax
18: eb 7f jmp 99 <crc16+0x99>
1a: c6 45 fb 00 movb $0x0,-0x5(%ebp)
1e: 8b 45 08 mov 0x8(%ebp),%eax
21: 90 nop
22: 90 nop
...
5c: 90 nop
5d: 90 nop
#   o \n
5e: 68 6f 0a 00 00 push $0xa6f
#   h e l l
63: 68 68 65 6c 6c push $0x6c6c6568
# arg 2: pointer to message to write,
# on the stack
68: 89 e1 mov %esp,%ecx
6a: ba 06 00 00 00 mov $0x6,%edx
# arg 1: file handle (stdout)
6f: bb 01 00 00 00 mov $0x1,%ebx
# system call number (sys_write)
74: b8 04 00 00 00 mov $0x4,%eax
# call kernel
79: cd 80 int $0x80
7b: eb 02 jmp 7f <crc16+0x7f>
7d: 27 daa
7e: 78 90 js 10 <crc16+0x10>
80: 90 nop
81: 90 nop
82: 90 nop
...
97: 90 nop
98: 90 nop
99: c9 leave
9a: c3 ret
```