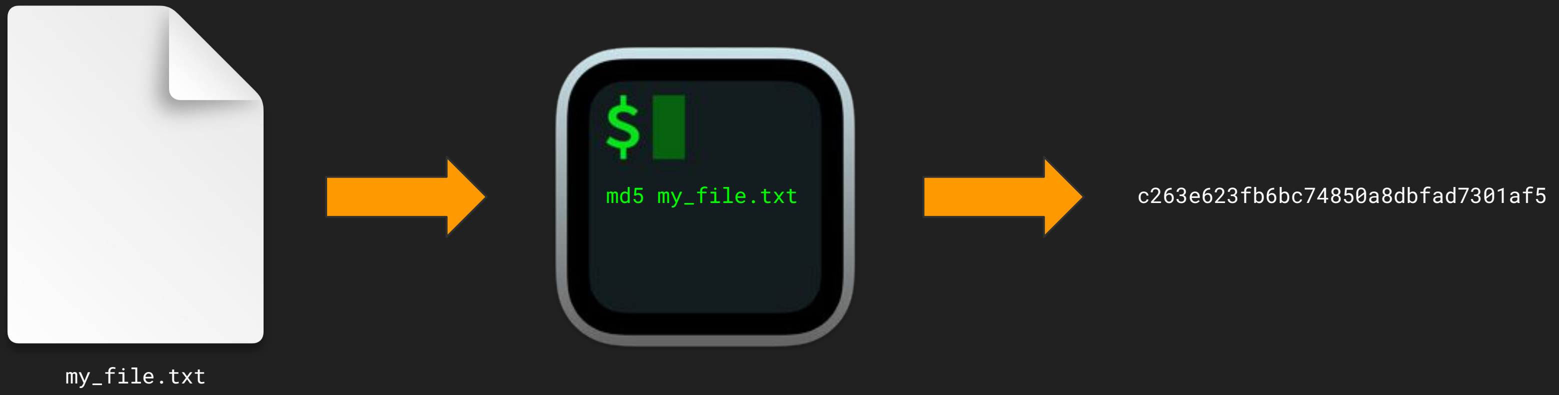
“Can we still use MD5?”

An md5 collision attack in practice

Simon Fransson

MD-what?

- “message-digest algorithm” v5
- Designed by MIT professor Ronald Rivest in 1992, when it was predicted MD4 would become insecure (spoiler alert, it did!).
- Computes a stable hash based on the contents of a file



```
$ md5 my_file.txt
```

```
c263e623fb6bc74850a8dbfad7301af5
```

Can I still use MD5?

It depends.

The MD5 message-digest algorithm is a **cryptographically broken** but **still widely used** hash function producing a 128-bit hash value.

Different types of attacks

- **Collision attack** 🔓 - find two different messages *m1* and *m2* such that hash(*m1*) = hash(*m2*).
- **Chosen-prefix attack** 🔓 - given two different prefixes *p1* and *p2*, find two appendages *m1* and *m2* such that hash(*p1* // *m1*) = hash(*p2* // *m2*),
- **Pre-image attack** 🔐 - find a message *m* such that it produces an already known hash *h*, hash(*m*) = *h*

Super simple C program

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

Looking at our executable

Looking at the compiled executable we can see the relevant byte code is prepended with a bunch of NULL bytes (macOS + x64, may vary by architecture):

```
gcc -o executable source.c
xxd executable | grep -B 50 Hello
00003ca0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003cb0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003cc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003cd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003ce0: 5548 89e5 4883 ec10 897d fc48 8975 f048  UH..H....}.H.u.H
…
00003f20: 00e9 aaff ffff 4865 6c6c 6f20 776f 726c  ......Hello worl
```

This is something we can exploit!

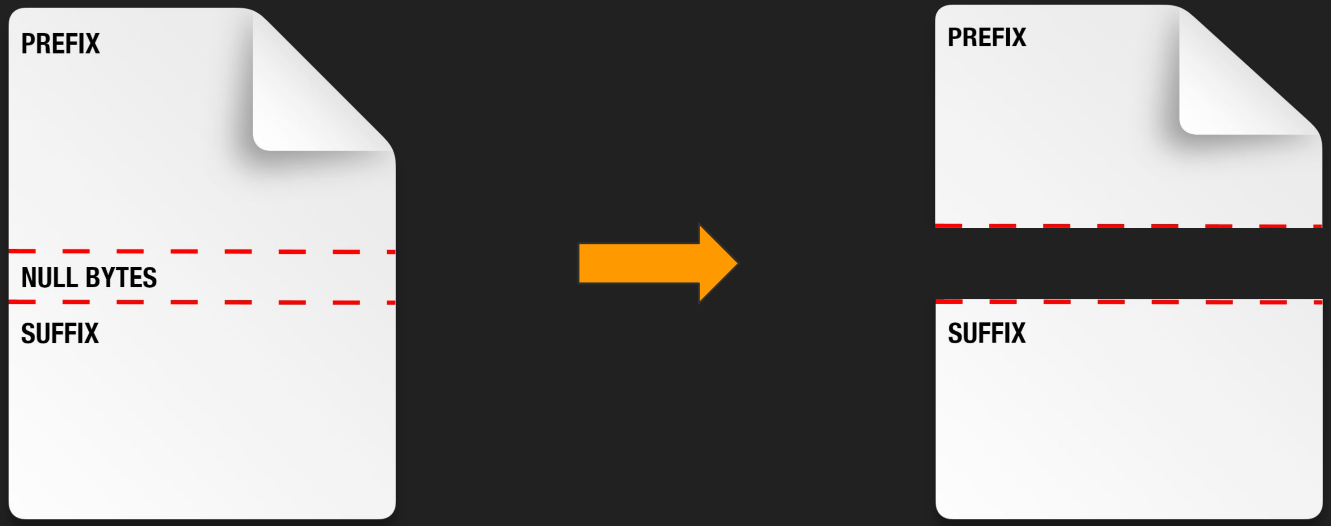
hashclash

Project HashClash - MD5 & SHA-1 cryptanalysis

> “Project HashClash is a Framework for MD5 & SHA-1 Differential Path Construction and Chosen-Prefix Collisions for MD5. It's goal is to further understanding and study of the weaknesses of MD5 and SHA-1.”

Comes with a tool called `fastcoll` - fast MD5 collision generator.

Split executable into prefix and suffix



Split executable into prefix and suffix

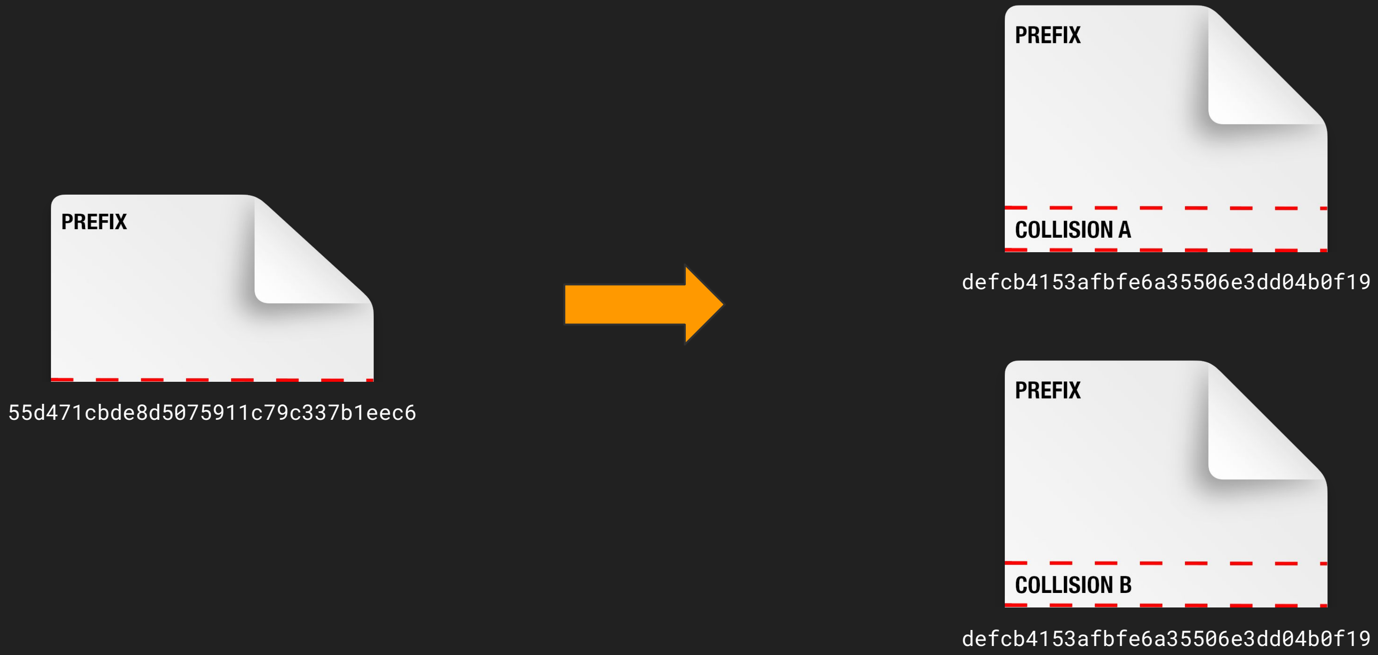
Collision appendage is typically 128 (0x80) prepended with 32 (0x20) empty bytes, so let's find 160 unused bytes and use that to split our executable in two parts - **prefix** and **suffix.**

```
# addr       coll   pad    result
0x00003ce0 - 0x80 - 0x20 = 0x00003c40

dd if=executable bs=1 status=none count=$((0x00003c40)) of=prefix
dd if=executable bs=1 status=none skip=$((0x00003ce0)) of=suffix
```

Different prefixes, same MD5 hash

```
md5_fastcoll -p prefix -o prefix_a prefix_b
```



PREFIX

55d471cbde8d5075911c79c337b1eec6

PREFIX

COLLISION A

defcb4153afbfe6a35506e3dd04b0f19

PREFIX

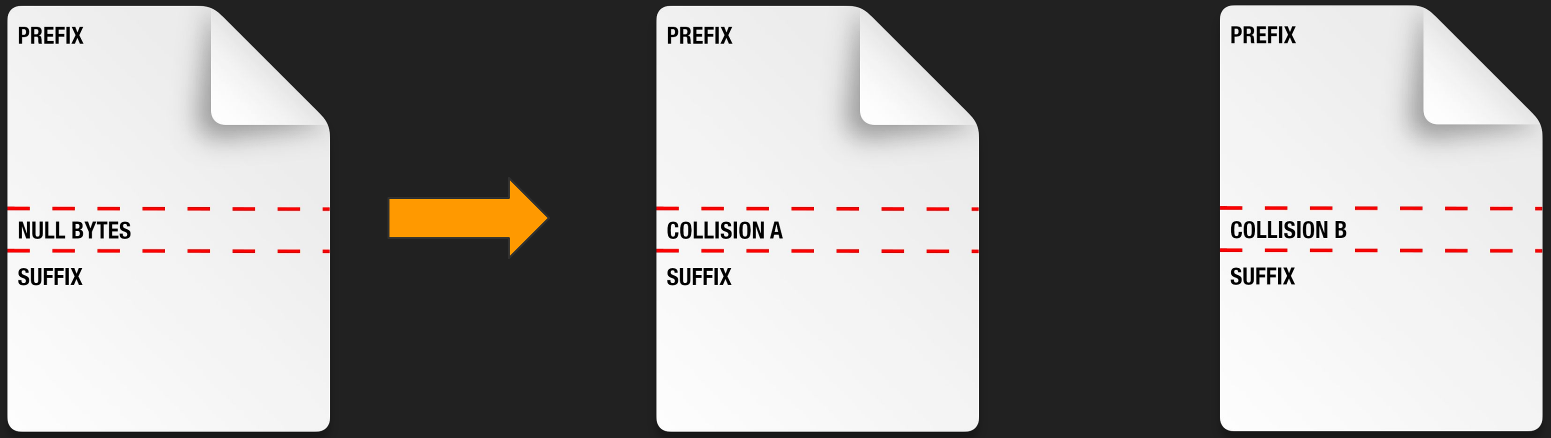
COLLISION B

defcb4153afbfe6a35506e3dd04b0f19

Join prefix and suffix

```
cat prefix_a + suffix > good

cat prefix_b + suffix > evil
```



```
./executable
bf1e84d300d3d505573fb5527b17f53b
```

```
./good
87162d02edadd36d8bae876ac36d0642
```

```
./evil
87162d02edadd36d8bae876ac36d0642
```

Same program with a hidden payload

```c
#include <stdio.h>

int good(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}

int evil(int argc, char *argv[])
{
    // TODO: Be evil
}

int main(int argc, char *argv[])
{
    if (0) // TODO: Find a branch condition
    {
        return evil(argc, argv);
    }

    return good(argc, argv);
}
```

Condition for executing the evil payload

A diff between the two prefixes shows that they consistently differ in only a couple of bits!
We can use this to let the executable read itself and lookup the value at a specific position.

```
b1   57   e6   13   18   45   dc   83   66   5d   26   b8   9d   d2   1a   2f
ff   6a   ad   d8   9b   14   86   03   53   72   84   29   ef   95   c5   fd    # 11011000
60   c3   ed   d7   6c   ec   dc   1d   0b   ba   6d   3b   ea   c3   27   67    # 11000011 / 00100111
52   47   3e   60   b9   b2   4f   fa   14   0a   01   8b   a8   16   d9   8b    # 10001011
ff   6a   ad   58   9b   14   86   03   53   72   84   29   ef   95   c5   fd    # 01011000
60   c3   ed   d7   6c   ec   dc   1d   0b   ba   6d   3b   ea   43   28   67    # 01000011 / 00101000
52   47   3e   60   b9   b2   4f   fa   14   0a   01   0b   a8   16   d9   8b    # 00001011
3c   aa   89   1a   bf   a9   ff   95   ec   2c   a1   93   36   7e   4f   e7
5a   14   19   35   94   8f   40   3e   28   d9   7c   86   4e   fc   f4   ce    # 00110101
25   c7   76   a5   ea   97   89   67   f2   59   b9   89   87   bc   e9   5c    # 10111100
e1   8c   0f   7b   6e   c6   43   d8   c9   60   37   fc   61   fb   f3   47    # 11111100
5a   14   19   b5   94   8f   40   3e   28   d9   7c   86   4e   fc   f4   ce    # 10110101
25   c7   76   a5   ea   97   89   67   f2   59   b9   89   87   3c   e9   5c    # 00111100
e1   8c   0f   7b   6e   c6   43   d8   c9   60   37   7c   61   fb   f3   47    # 01111100
00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
```

Payload branching added

```c
#include <stdio.h>

int good(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}

int evil(int argc, char *argv[])
{
    // TODO: Be evil
}

int main(int argc, char *argv[])
{
    if (read_byte(0x00000693) == 0xff) // TODO: How do we know which value to test against?
    {
        return evil(argc, argv);
    }

    return good(argc, argv);
}
```

Toggle evil payload

We can’t just lookup the expected byte value and recompile + run fastcoll, as the value will change with each new build. But we can lookup the value and modify the compiled binary, or use a recompiled suffix part (which is easier than finding that one byte in a… byte stack?).

```
BYTE=$(xxd -s 0x00000693 -l 1 evil | awk '{ print $2 }')
sed -i '' 's/== 0xff/== 0x${BYTE}/' source.c
gcc -o executable source.c
dd if=executable bs=1 status=none skip=$((0x00000700)) of=suffix
cat prefix_col1 suffix > good
cat prefix_col2 suffix > evil
```

Demo time!

Implications

- Provide certain website visitors with a (seemingly) normal file and others with a malignant copy, both sharing the same MD5.
- Make adjustments to critical binaries (gcc etc.) on System A, while System B appears normal and they share the same MD5 (Reflections on Trusting Trust).
- Avoid MD5 (and SHA-1), SHA-2 is probably better

Thanks!

https://github.com/dessibelle/md5-collision