14:04 Flush+Reload

by Taylor Hornby

Dear Editors and Readers of PoC||GTFO,

You've been lied to about how your computer works. You see, in a programming class they teach you just enough for you to get on with your job and no more. What you learn is a mere abstraction of the very complicated piece of physics sitting under your desk. To use your computer to its fullest potential, you must forget the familiar abstraction and finally see your computer for what it really is. Come with me, as we take a small step towards enlightenment.

You know what makes a computer—or so you think. There is a processor. There is a bank of main memory, which the processor reads, writes, and executes from. And there are processes, those entities that from time to time get loaded into the processor to do their work.

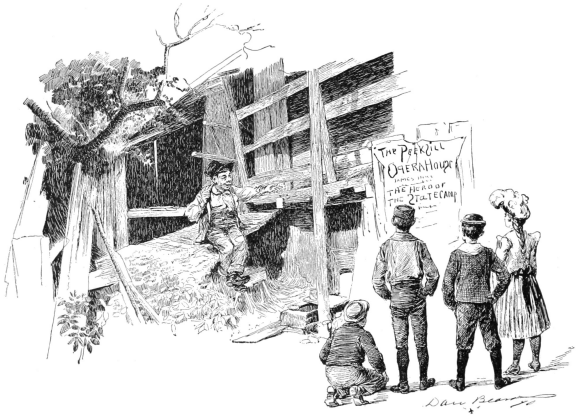
As we know, processes shouldn't be trusted to play well together, and need to be kept separate. Many of the processor's features were added to keep those processes isolated. It would be quite bad if one process could talk to another without the system administrator's permission.

We also know that the faster a computer is, the more work it can do and the more useful it is. Even more features were introduced to the processor in order to make it go as fast as possible.

Accordingly, your processor most likely has a memory cache sitting between main memory and the processor, remembering recently-read data and code, so that the next time the processor reads from the same address, it doesn't have to reach all the way out to main memory. The vendors will say this feature was added to make the processor go faster, and it does do a great job of that. But I will show you that the cache is *also* a feature to help hackers get around those annoying access controls that system administrators seem to love.

What I'm going to do is show you how to send a text message from one process to the other, using only memory *reads*. What!? How could this be possible? According to your programming class, you say, reads from memory are just reads, they can't be used to send messages!

[9] Usenix Security 2014



The gist is this: the cache remembers recently executed code, which means that it must also remember *which* code was recently executed. Processes are in control of the code they execute, so what we can do is execute a special pattern of code that the cache will remember. When the second process gets a chance to run, it will read the pattern out of the cache and recover the message. Oh how thoughtful it was of the processor designers to add this feature!

The undocumented feature we'll be using is called "Flush+Reload," and it was originally discovered by Yuval Yarom and Katrina Falkner.[9] It's available in most modern Intel processors, so if you've got one of those, you should be able to follow along.

It works like this. When Sally the Sender process gets loaded into memory, one copy of all her executed code gets loaded into main memory. When Robert the Receiver process loads Sally's binary into his address space, the operating system isn't going to load a second copy: that would be wasteful. Instead, it's just going to point Robert's page tables at Sally's memory. If Sally and Robert could both write to the memory, it would be a huge problem since they could simply talk by writing messages to each other in the shared memory. But that isn't a problem, because one of those processor security features stops both Sally and Robert from being able to write to the memory. How do they communicate then?

When Sally the Sender executes some of her code, the cache—the last-level cache, to be specific—is going to remember her most recently executed code. When Robert the Receiver reads a chunk of code in Sally's binary, the read operation is going to be sent through the very same cache. So: if Sally ran the code not too long ago, Robert's read will happen very fast. If Sally hasn't run the code in a while, Robert's read is going to be slow.

Sally and Robert are going to agree ahead of time on 27 locations in Sally's binary. That's one location for each letter of the alphabet, and one left over for the space character. To send a message to Robert, Sally is going to spell out the message by executing the code at the location for the letter she wants to send. Robert is going to continually read from all 27 locations in a loop, and when one of them happens faster than usual, he'll know that's a letter Sally just sent.

Figure 1 contains the source code for Sally's binary. Notice that it doesn't even explicitly make any system calls.

This program takes a message to send on the command-line and simply passes the processor's thread of execution over the probe site corresponding to that character. To have Sally send the message "`THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG`" we just compile it without optimizations, then run it.

But how does Robert receive the message? Robert runs the program whose source code is at `flush-reload/myversion`. The key to that program is this bit of code, which times how long it takes to read from an address, and then flushes it from the cache.

```
__attribute__((always_inline))
inline unsigned long probe(char *adrs){
  volatile unsigned long time;

  asm __volatile__ (
    "  mfence             \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  lfence             \n"
    "  movl %%eax, %%esi  \n"
    "  movl (%1), %%eax   \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  subl %%esi, %%eax  \n"
    "  clflush 0(%1)      \n"
    : "=a" (time)
    : "c" (adrs)
    :  "%esi", "%edx");
  return time;
}
```

By repeatedly running this code on those special probe sites in Sally's binary, Robert will see which letters Sally is sending. Robert just needs to know where those probe sites are. It's a matter of filtering the output of objdump to find those addresses, which can be done with this handy script:

```
#!/bin/bash
for letter in {A..Z}
do
    addr=$(objdump -D -M intel msg       | \
           sed -n -e "/<$letter>/,\$p" | \
           grep call | head -n 1        | \
           cut -d ':' -f 1 | tr -d ' ');
    echo -n "-p $letter:0x$addr "
done
addr=$(objdump -D -M intel msg          | \
        sed -n -e "/<SP>/,\$p"          | \
        grep call | head -n 1           | \
        cut -d ':' -f 1 | tr -d ' ');
echo "-p _:0x$addr"
```

Assuming this script works, it will output a list of command-line arguments for the receiver, enumerating which addresses to watch for getting entered into the cache:

```
-p A:0x407cc5 -p B:0x416cd5 -p C:0x425ce5
-p D:0x434cf5 -p E:0x443d05 -p F:0x452d15
-p G:0x461d25 -p H:0x470d35 -p I:0x47fd45
-p J:0x48ed55 -p K:0x49dd65 -p L:0x4acd75
-p M:0x4bbd85 -p N:0x4cad95 -p O:0x4d9da5
-p P:0x4e8db5 -p Q:0x4f7dc5 -p R:0x506dd5
-p S:0x515de5 -p T:0x524df5 -p U:0x533e05
-p V:0x542e15 -p W:0x551e25 -p X:0x560e35
-p Y:0x56fe45 -p Z:0x57ee55 -p _:0x58de65
```

```
/* msg.c - Send a message through the Flush+Reload cache side-channel.
 * Written Taylor Hornby for PoC||GTFO 0x14.
 */

// We surround the probe sites with padding. This makes sure they're in
// different page frames which reduces noise from prefetching, etc.
unsigned int padding = 0;
#define PADDING_A padding += 1;
#define PADDING_B PADDING_A PADDING_A
#define PADDING_C PADDING_B PADDING_B
#define PADDING_D PADDING_C PADDING_C
#define PADDING_E PADDING_D PADDING_D
#define PADDING_F PADDING_E PADDING_E
#define PADDING_G PADDING_F PADDING_F
#define PADDING_H PADDING_G PADDING_G
#define PADDING_I PADDING_H PADDING_H
#define PADDING_J PADDING_I PADDING_I
#define PADDING   PADDING_J PADDING_J
#define PADDING_K PADDING_K PADDING_K

// The probe sites will be call instructions to this empty function. It
// doesn't have to be a call instruction; it's just easy to grep for.
void null() { }
#define PROBE null();

// One probe site for each letter of the alphabet and space.
void A() { PADDING PROBE PADDING } void B() { PADDING PROBE PADDING }
void C() { PADDING PROBE PADDING } void D() { PADDING PROBE PADDING }
void E() { PADDING PROBE PADDING } void F() { PADDING PROBE PADDING }
void G() { PADDING PROBE PADDING } void H() { PADDING PROBE PADDING }
void I() { PADDING PROBE PADDING } void J() { PADDING PROBE PADDING }
void K() { PADDING PROBE PADDING } void L() { PADDING PROBE PADDING }
void M() { PADDING PROBE PADDING } void N() { PADDING PROBE PADDING }
void O() { PADDING PROBE PADDING } void P() { PADDING PROBE PADDING }
void Q() { PADDING PROBE PADDING } void R() { PADDING PROBE PADDING }
void S() { PADDING PROBE PADDING } void T() { PADDING PROBE PADDING }
void U() { PADDING PROBE PADDING } void V() { PADDING PROBE PADDING }
void W() { PADDING PROBE PADDING } void X() { PADDING PROBE PADDING }
void Y() { PADDING PROBE PADDING } void Z() { PADDING PROBE PADDING }
void SP() { PADDING PROBE PADDING }

int main(int argc, char **argv){
    char *p;
    char lowercase;

    if (argc != 2)
        return 1;

    for (p = argv[1]; *p != 0; ++p) {
        // Execute the probe corresponding to the letter to send.
        lowercase = *p | 32;
        switch(lowercase) {
            case 'a': A(); break; case 'b': B(); break;
            case 'c': C(); break; case 'd': D(); break;
            case 'e': E(); break; case 'f': F(); break;
            case 'g': G(); break; case 'h': H(); break;
            case 'i': I(); break; case 'j': J(); break;
            case 'k': K(); break; case 'l': L(); break;
            case 'm': M(); break; case 'n': N(); break;
            case 'o': O(); break; case 'p': P(); break;
            case 'q': Q(); break; case 'r': R(); break;
            case 's': S(); break; case 't': T(); break;
            case 'u': U(); break; case 'v': V(); break;
            case 'w': W(); break; case 'x': X(); break;
            case 'y': Y(); break; case 'z': Z(); break;
            case ' ': SP(); break;
        }
    }

    return 0;
}
```

Figure 1. Sally's Executable

The letter before the colon is the name of the probe site, followed by the address to watch after the colon. With those addresses, Robert can run the tool and receive Sally's messages.

```
$ ./spy -e ./msg -t 120 -s 20000 \
-p A:0x407cc5 -p B:0x416cd5 -p C:0x425ce5 \
-p D:0x434cf5 -p E:0x443d05 -p F:0x452d15 \
-p G:0x461d25 -p H:0x470d35 -p I:0x47fd45 \
-p J:0x48ed55 -p K:0x49dd65 -p L:0x4acd75 \
-p M:0x4bbd85 -p N:0x4cad95 -p O:0x4d9da5 \
-p P:0x4e8db5 -p Q:0x4f7dc5 -p R:0x506dd5 \
-p S:0x515de5 -p T:0x524df5 -p U:0x533e05 \
-p V:0x542e15 -p W:0x551e25 -p X:0x560e35 \
-p Y:0x56fe45 -p Z:0x57ee55 -p _:0x58de65
```

The `-e` option is the path to Sally's binary, which must be exactly the same path as Sally executes. The `-t` parameter is the threshold that decides what's a fast access or not. If the memory read is faster than that many clock cycles, it will be considered fast, which is to say that it's in the cache. The `-s` option is how often in clock cycles to check all of the probes.

With Robert now listening for Sally's messages, Sally can run this command in another terminal as another user to transmit her message.

```
$ ./msg "The quick brown fox jumps over the
    lazy dog"
```

```
WARNING: This processor does not have an
    invariant TSC.
Detected ELF type: Executable.
T|H|E|_|Q|U|I|C|K|_|_|B|B|R|O|W|N|_|F|O|X|_|
    J|U|M|P|S|_|O|V|E|R|_|T|H|E|_|L|A|Z|Y|_|
    D|O|G|
```

There's a bit of noise in the signal (note the replicated `B`'s), but it works! Don't take my word for it, try it for yourself! It's an eerie feeling to see one process send a message to another even though all they're doing is reading from memory.

Now you see what the cache really is. Not only does it make your computer go faster, it also has this handy feature that lets you send messages between processes without having to go through a system call. You're one step closer to enlightenment.

This is just the beginning. You'll find a collection of tools and experiments that go much further than this.[10] The attacks there use Flush+Reload to find out which PDF file you've opened, which web pages you're visiting, and more.

I leave two open challenges to you fine readers:

1. Make the message-sending tool reliable, so that it doesn't mangle messages even a little bit. Even cooler would be to make it a two-way reliable chat.

2. Extend the PDF-distinguishing attack in my poppler experiment[11] to determine which page of `pocorgtfo14.pdf` is being viewed. As I'm reading this issue of PoC||GTFO, I want you to be able to tell which page I'm looking at through the side channel.

Best of luck!
—Taylor Hornby

[10] `git clone https://github.com/defuse/flush-reload-attacks`
[11] `experiments/poppler`