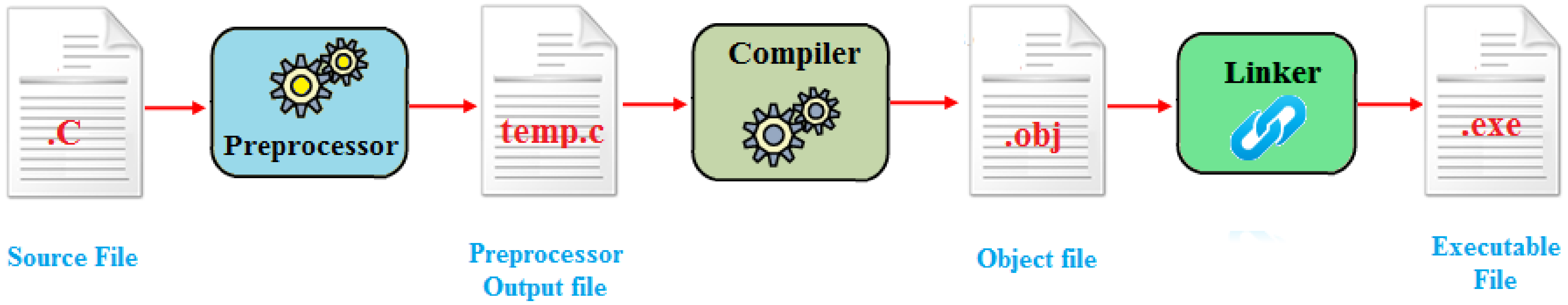
Spying with LD_PRELOAD

Author: Russell Francis (rfrancis@codelogic.com)

Git: https://github.com/russellfrancis/ld_preload_spy

Compilation & Linking

- Applications are compiled from source code into object files using a compiler.
- The linker takes several object files and "links" them into a single executable file.
- The linker can link an application either "statically" or "dynamically".



Static vs Dynamic Linking

Static Linking

- All referenced libraries are copied into the produced executable.
- The executable does not need to resolve any references to externally defined methods at runtime because they are all included inside the executable!
- A static executable is larger than a dynamic executable because it includes everything needed to run.
- If a bug is identified and fixed or introduced in a dependent library. The change will not be reflected in the statically linked application.

Dynamic Linking

- A dynamically linked executable includes only the function signatures which are needed to resolve used functions at runtime.
- The functions must be resolved at runtime, if they can't be found the application will fail.
- A dynamic executable is smaller because it doesn't include all of the code need to run.
- If a dynamic library is updated, all dynamically linked applications will immediately start using the new version without needing to be rebuilt.
- Dynamic linking is very common because it allows more easily updating software and keeps the size of the generated executables small and allows more efficient memory usage by the OS.

What is LD_PRELOAD?

LD_PRELOAD is an environment variable which modifies the behavior of shared library resolution when executing a dynamically linked executable.

It allows us to add our own methods which can override behavior defined in other shared libraries.

LD_PRELOAD will only work on Linux and other UNIX-like systems.

OS X has a similar mechanism `DYLD_INSERT_LIBRARIES="./test.dylib" DYLD_FORCE_FLAT_NAMESPACE=1 application`

Windows has a similar mechanism where you can provide a library in a registry key "AppInit_DLLs" which gets included in every process!

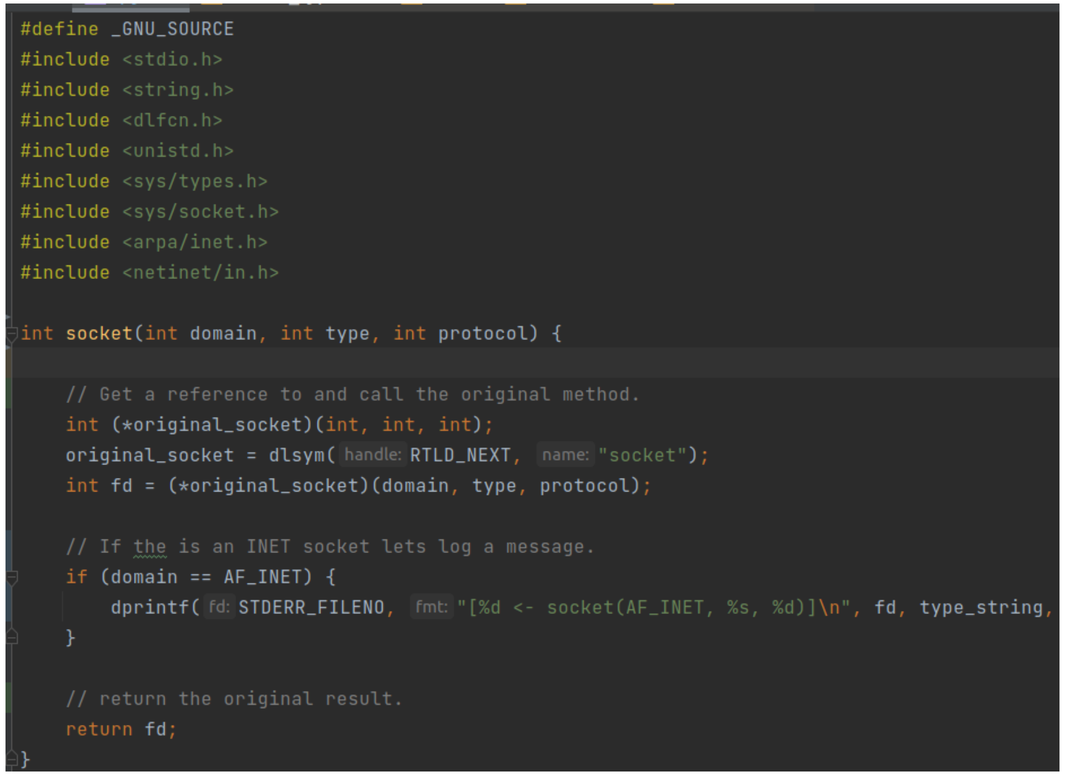
Can we do anything useful with this?

- We could override some functions in libc to identify all of the files that an application interacts with on the local system.

- Override getaddrinfo() to identify all of the DNS names which an application resolves.

- Override socket() to see what IP addresses an application connects to.

An example socket()



```c
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int socket(int domain, int type, int protocol) {

    // Get a reference to and call the original method.
    int (*original_socket)(int, int, int);
    original_socket = dlsym( handle: RTLD_NEXT,  name: "socket");
    int fd = (*original_socket)(domain, type, protocol);

    // If the is an INET socket lets log a message.
    if (domain == AF_INET) {
        dprintf( fd: STDERR_FILENO,  fmt: "[%d <- socket(AF_INET, %s, %d)]\n", fd, type_string,
    }

    // return the original result.
    return fd;
}
```

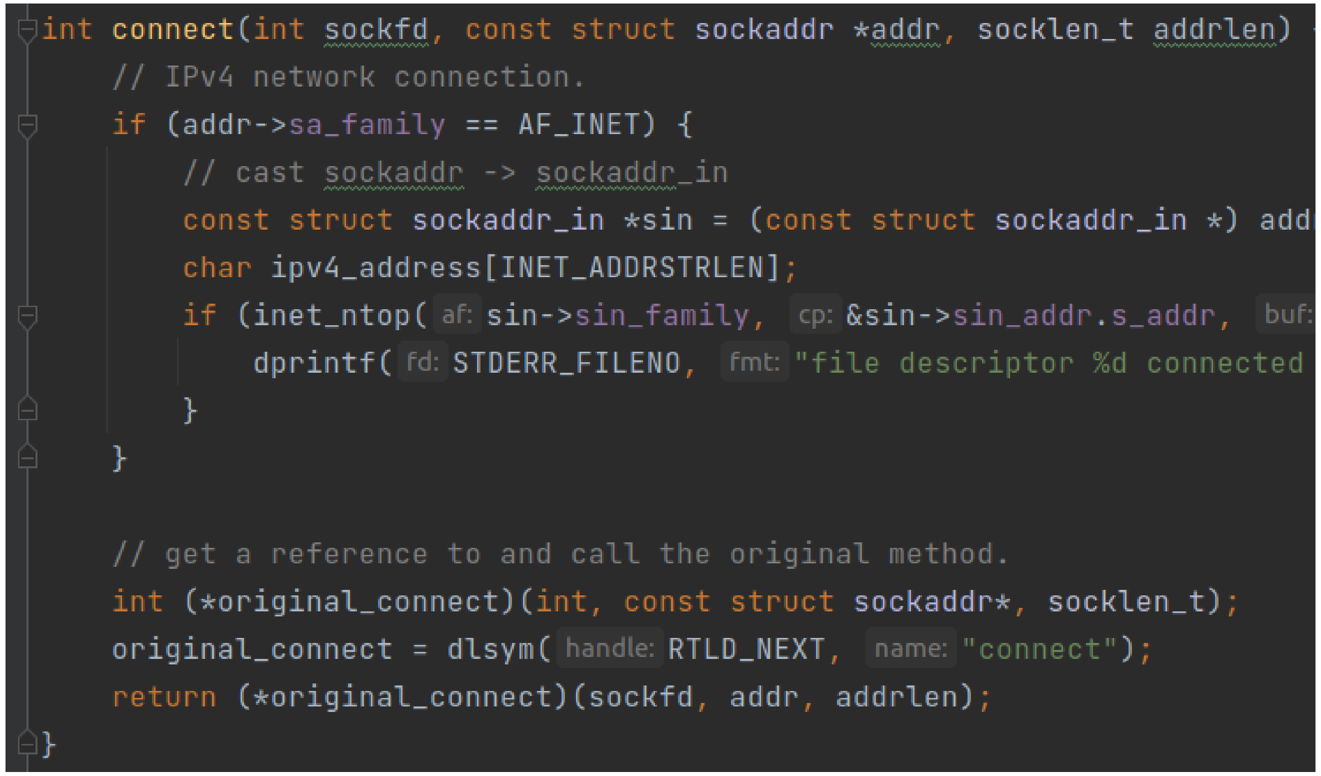
- Lets see if we can do this by overriding the `socket()` call which is a prerequisite for any network communication.

- # git checkout demo_1
- # make
- # LD_PRELOAD=./spy.so curl -s -S -k
- -X GET http://www.google.com > /dev/null
- [5 <- socket(AF_INET, 1, 6)]

- This indicates what curl called our overridden socket function and create file descriptor 5 to communicate with it.

- Is there anything else we can do?

Lets connect()

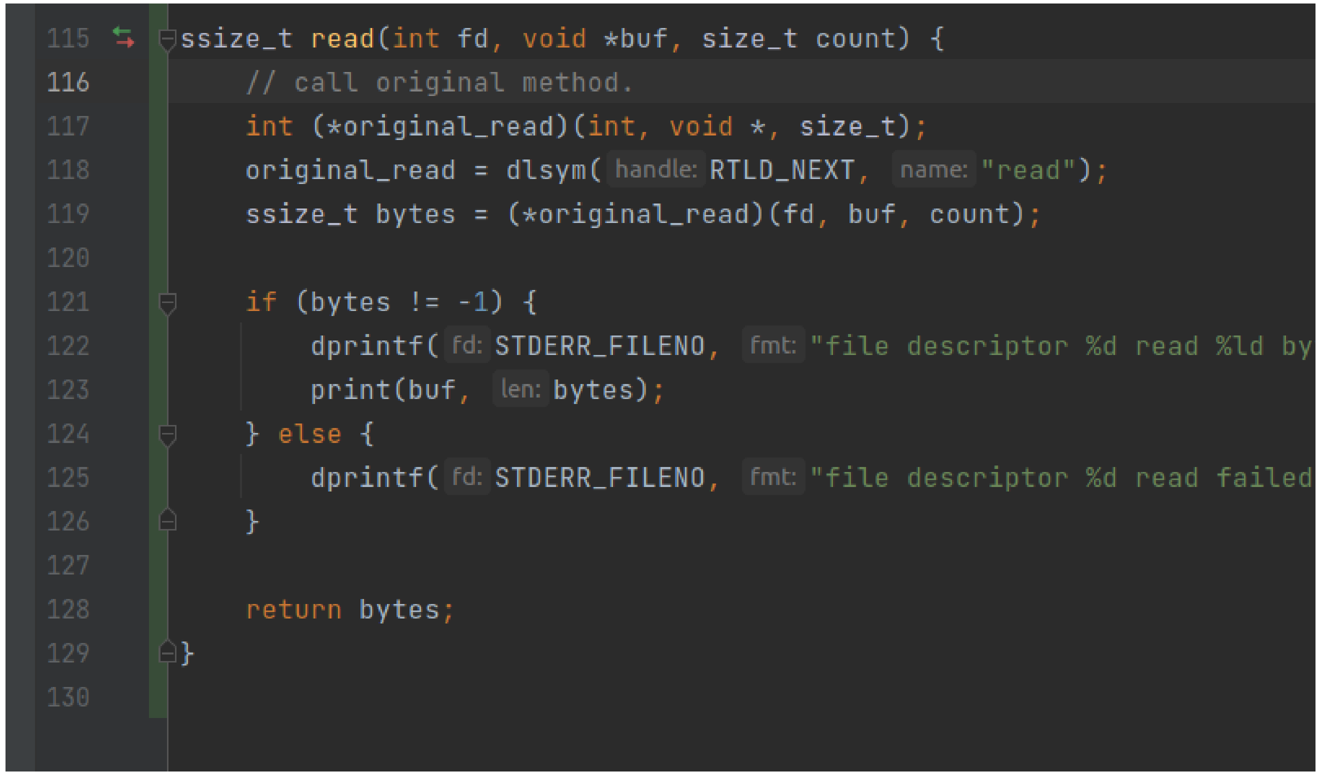


```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
    // IPv4 network connection.
    if (addr->sa_family == AF_INET) {
        // cast sockaddr -> sockaddr_in
        const struct sockaddr_in *sin = (const struct sockaddr_in *) add
        char ipv4_address[INET_ADDRSTRLEN];
        if (inet_ntop( af: sin->sin_family,  cp: &sin->sin_addr.s_addr,  buf:
            dprintf( fd: STDERR_FILENO,  fmt: "file descriptor %d connected
        }
    }

    // get a reference to and call the original method.
    int (*original_connect)(int, const struct sockaddr*, socklen_t);
    original_connect = dlsym( handle: RTLD_NEXT,  name: "connect");
    return (*original_connect)(sockfd, addr, addrlen);
}
```

- # git checkout demo_2
- # make
- # ./run.sh
- [5 <- socket(AF_INET, 1, 6)]
- file descriptor 5 connected to (address = 172.217.165.132, port = 20480).

- Neat, now we can see that the application created a socket with file descripter 5 and then bound that file descriptor to the IPv4 address 172.217.165.132:20480

read/write send/recv … why not?



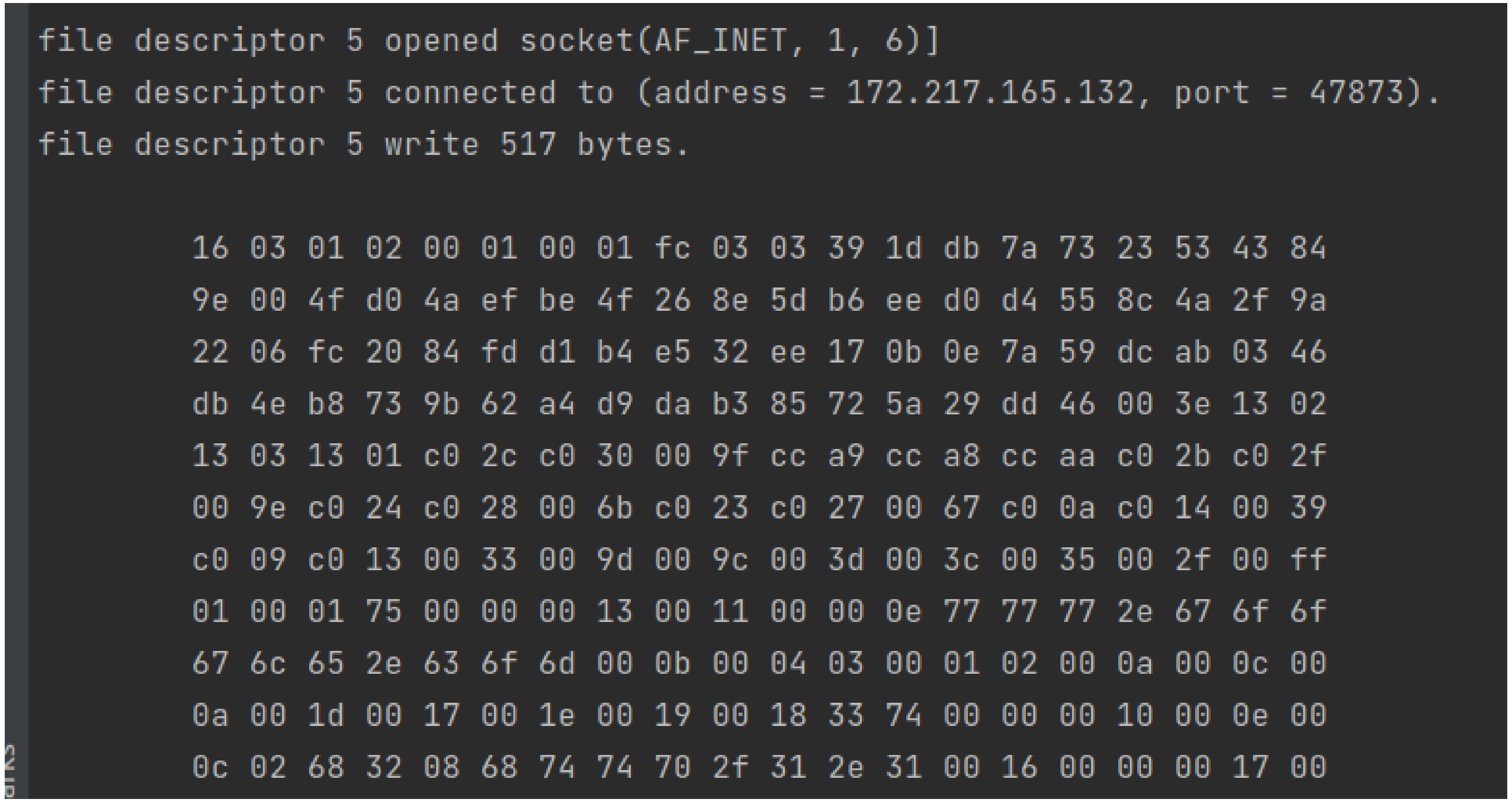
```
ssize_t read(int fd, void *buf, size_t count) {
    // call original method.
    int (*original_read)(int, void *, size_t);
    original_read = dlsym( handle: RTLD_NEXT,  name: "read");
    ssize_t bytes = (*original_read)(fd, buf, count);

    if (bytes != -1) {
        dprintf( fd: STDERR_FILENO,  fmt: "file descriptor %d read %ld by
        print(buf,  len: bytes);
    } else {
        dprintf( fd: STDERR_FILENO,  fmt: "file descriptor %d read failed
    }

    return bytes;
}
```

- # git checkout demo_3
- # make clean
- # make
- # ./run.sh

- Pretty neat, for HTTP requests we can tie together the file descriptor with the IP address and eavesdrop on reads and writes to that endpoint.

- This would allow us to extract the path as well as headers and request payloads from a running application.

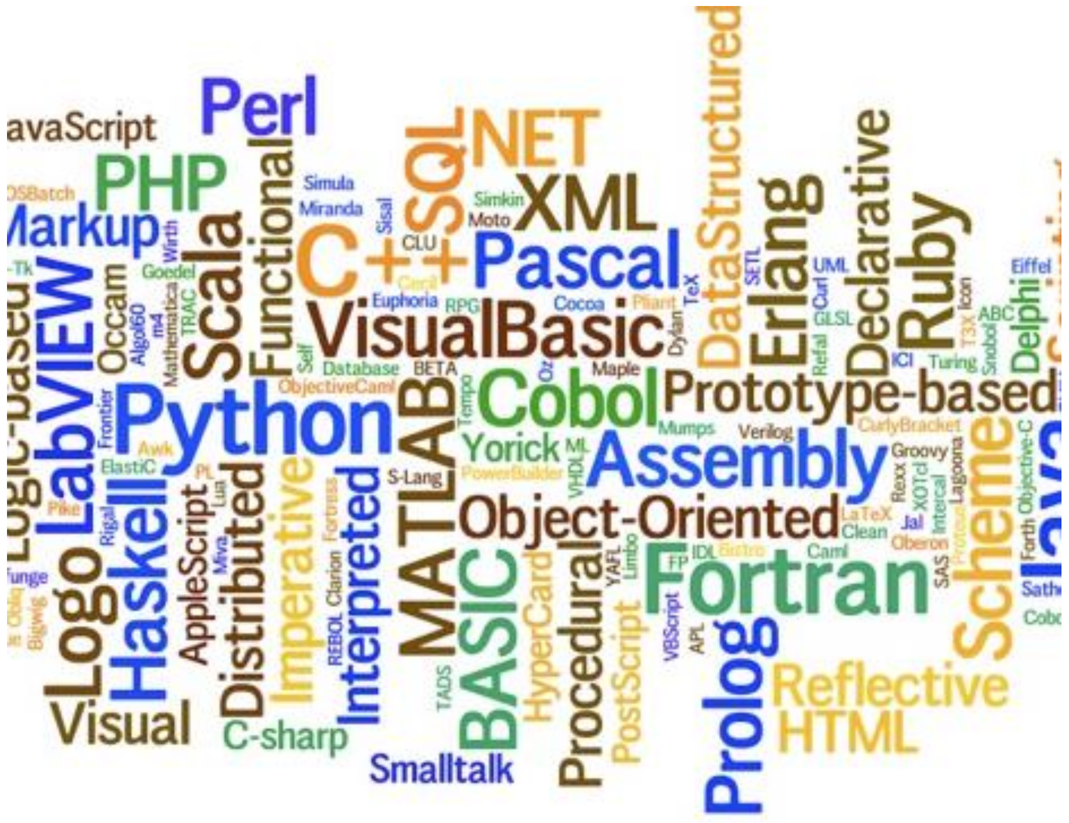
Does this work with SSL?



```
file descriptor 5 opened socket(AF_INET, 1, 6)]
file descriptor 5 connected to (address = 172.217.165.132, port = 47873).
file descriptor 5 write 517 bytes.


        16 03 01 02 00 01 00 01 fc 03 03 39 1d db 7a 73 23 53 43 84
        9e 00 4f d0 4a ef be 4f 26 8e 5d b6 ee d0 d4 55 8c 4a 2f 9a
        22 06 fc 20 84 fd d1 b4 e5 32 ee 17 0b 0e 7a 59 dc ab 03 46
        db 4e b8 73 9b 62 a4 d9 da b3 85 72 5a 29 dd 46 00 3e 13 02
        13 03 13 01 c0 2c c0 30 00 9f cc a9 cc a8 cc aa c0 2b c0 2f
        00 9e c0 24 c0 28 00 6b c0 23 c0 27 00 67 c0 0a c0 14 00 39
        c0 09 c0 13 00 33 00 9d 00 9c 00 3d 00 3c 00 35 00 2f 00 ff
        01 00 01 75 00 00 00 13 00 11 00 00 0e 77 77 77 2e 67 6f 6f
        67 6c 65 2e 63 6f 6d 00 0b 00 04 03 00 01 02 00 0a 00 0c 00
        0a 00 1d 00 17 00 1e 00 19 00 18 33 74 00 00 00 10 00 0e 00
        0c 02 68 32 08 68 74 74 70 2f 31 2e 31 00 16 00 00 00 17 00
```

Sad Panda

But it works almost anywhere!



This technique works on any dynamically linked application! It doesn't matter what language was used whether (C, C++, Ruby, Java, Python, Rust, Javascript, …) If it runs as a dynamically linked executable on a Linux system it is ultimately using libc to make network calls and we can spy on it.

It works regardless of application role or frameworks used!

- While we traveled down the "client" path, the same technique could be used on a server application to determine what hosts and ports it was listening on and glean information about who if anyone connected from where and what they requested.

Detailed Information on Interconnectedness

Detailed information on the high level interconnected-ness of applications, not much detail on the internal structure of an application.

Conclusion

Hopefully I've piqued your interest, there are lots of fun things you can do to get a better idea of what your applications are doing using this technique.