

Practical Binary Analysis #11

Seminar @ Gondow Lab.

What you will learn

1. Internals of `libdft`
 - a. data structure of `libdft`
 - b. how `libdft` works
2. How to use `libdft` to build DTA-tools
 - a. a tool that prevents remoto control-hijacking attacks
 - b. a tool that automatically detects information leaks

Overview

1. Internals of `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circumventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Overview

1. Internals of `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circumventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

What is `libdft`

- Open source binary-level taint tracking library
- Byte-granularity taint-tracking system built on Intel Pin
- Supports only 32-bit x86
 - although you can use it on a 64-bit platform
- Relies on legacy versions of Pin
- Supports only for "Regular" x86 instructions,
 - not for extended instruction sets like MMX or SSE.



taint : the effect of something bad or unpleasant.

`libdft` is based on Pin (between 2.11 and 2.13)

64-bit version of `libdft` : <https://github.com/AngoraFuzzer/libdft64>

Overview

1. Internals of `libdft`

- Shadow Memory
 - how to store taint info
- Virtual CPU
 - how to propagate taint info
- The libdft API and I/O interface
 - how to instrument

2. Using DTA to Detect Remote Control-Hijacking

3. Circumventing DTA with implicit Flows

4. A DTA-Based Data Exfiltration Detector

Internals of libdft - Shadow Memory

Bitmap (1 color)

- Supports only 1 taint color.
- Slightly faster and use less memory than STAB.

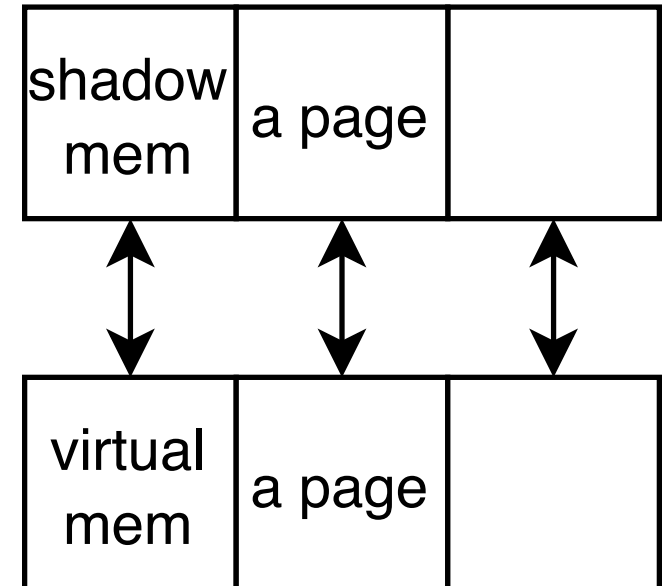
Bitmap (1 color)

1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

Internals of libdft - Shadow Memory

STAB : Segment Translation Table (8 color)

- Supports 8 taint color (because we use 8 bits to contain color).
- Contains one entry for every memory page.
- Shadow memory is allocated in page-sized chunks.
- Input and Output of STAB
 - input : some upper bits of virtual memory address
 - 16 bit if each page size is 2^{16} B(= 64KB)
 - output : some upper bits of shadow memory address
 - 16 bit if each page size is 2^{16} B(= 64KB)
- Lower bits can be used to access each shadow memory.
- Shadow memory pages is adjacent if the corresponding virtual memory is adjacent.



Internals of libdft - Virtual CPU

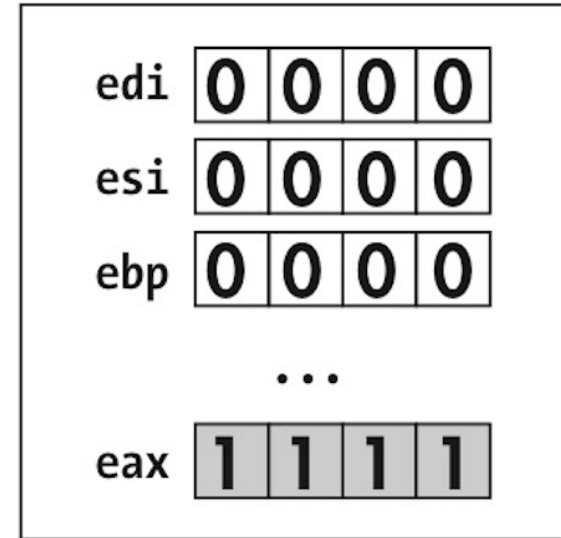
- *Virtual CPU* keeps track of the taint status of CPU register.
 - This is stored in memory as a special data structure.
- Virtual CPU is a kind of shadow memory.
 - for each of 32-bit general-purpose CPU registers.



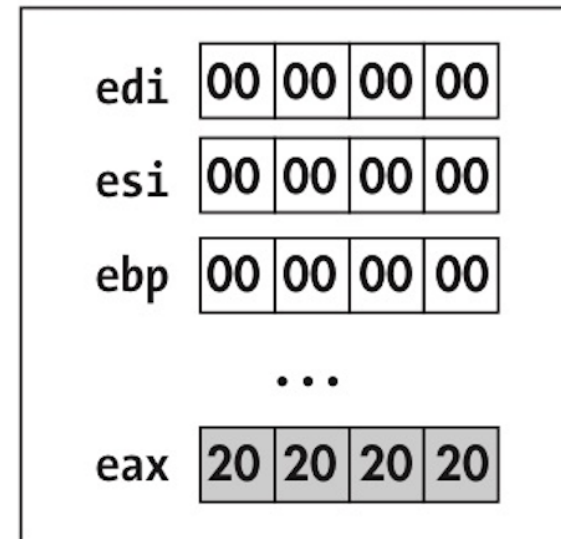
32-bit general purpose register of x86: edi, esi, ebp, esp, ebx, edx, ecx, and eax

Virtual CPU

VCPU (1 color)



VCPU (8 colors)



Internals of libdft - libdft API and I/O interface

- `libdft` provides a taint tracking API.
- Two import tools for building DTA tools is those that
 - manipulate tagmap (Tagmap API)
 - add callbacks and instrument code

Internals of libdft - libdft API and I/O interface

- `libdft` provides a taint tracking API.
- Two import tools for building DTA tools is those that
 - manipulate tagmap (Tagmap API) ←
 - add callbacks and instrument code

Tagmap API

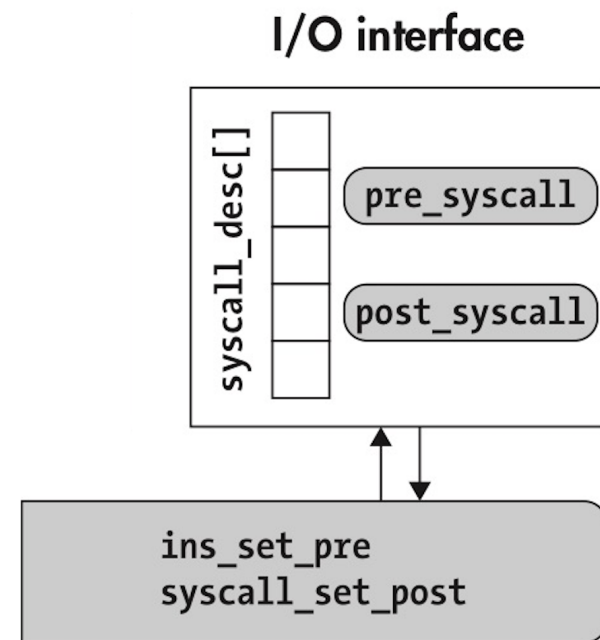
- `tagmap_setb()` : sets status of tagmap in byte granularity.
 - `tagmap_setn()` : taints arbitrary number of bytes.
- `tagmap_getb()` : gets status of tagmap in byte granularity.
 - `tagmap_getn()` : checks arbitrary number of bytes.

libdft API and I/O interface

- `libdft` provides a taint tracking API.
- Two import tools for building DTA tools is those that
 - `manipulate tagmap` (Tagmap API)
 - add callbacks and instrument code ←

API for adding callbacks and instrumentation code

- `syscall_set_pre()` : register callbacks for syscall events.
- `syscall_set_post()` : same as above
- `syscall_desc[]` : store syscall pre- and post-handlers.
 - use *syscall number* to index this array.
- `ins_set_pre` : register callbacks for instructions.
- `ins_set_post` : same as above



Taint Policy

- `libdft` taint policy defines the following classes of instructions.
 - Each of these classes define how to propagate and merge taint.
1. ALU : arithmetic and logic instruction such as `add`, `sbb`, `and`, `xor`, `div`, and `imul`
 2. XFER : instructions that copy a value such as `mov`. Simply copies the taint info.
 3. CLR : (=clear), instructions that reset taint info of output operands
 4. SPECIAL : instructions that require special rules
 5. FPU, MMX, SSE : instructions `libdft` doesn't support. Doesn't propagate taint info, causing undertainting.

Overview

1. introducing `libdft`
2. Using DTA to Detect Remote Control-Hijacking
 - Examples of `exec` family
 - Header files
 - Functions
 - `main` function
 - Details of funcs
 - Test of Control-Flow Hijacking
3. Circumventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Remote Control-Hijacking Attack

- Goal is to detect attacks where data received from network is used to control the argument of `execve` call (-> arbitrary code execution).
- taint source : the network receive functions, `recv` and `recvfrom`.
- taint sink : `execve`

Examples of **exec** family

	引数の渡し方(l or v)	環境変数(e)	パス検索(p)
exec <u>l</u>	list	引き継ぐ	しない
exec <u>v</u>	vector	引き継ぐ	しない
exec <u>e</u>	list	渡す	しない
exec <u>ve</u>	vector	渡す	しない
exec <u>lp</u>	list	引き継ぐ	する

```
#include <unistd.h>
int main(void){
    char *argv[] = {"ls", "-l", NULL};
    char *envp[] = {NULL};
    execve("/bin/ls", argv, envp);
}
```


Header files

```
// dta-execve.cpp
...
#include "pin.H"           // libdft uses Pin

#include "branch_pred.h"   // hints for branch prediction
#include "libdft_api.h"    // libdft base API
#include "syscall_desc.h"
#include "tagmap.h"
...
```

- all `libdft` tools are just Pin tools linked with the `libdft` library.

Functions

```
extern syscall_desc_t syscall_desc[SYSCALL_MAX];           // to store syscall hooks

void alert(uintptr_t addr, const char *source, uint8_t tag); // print alert and exit
void check_string_taint(const char *str, const char *source); // check whether tainted
static post_socketcall_hook(syscall_ctx_t *ctx);           // taint source
static pre_execve_hook(syscall_ctx_t *ctx);                // taint sink
```

- `syscall_desc` : Index this array with `syscall` number of `syscall` you're installing
 - such as `__NR_socketcall` or `__NR_execve`.
- Details of these functions will be explained later.

main function

```
int main(int argc, char **argv) {
    PIN_InitSymbols();

    if (unlikely(PIN_Init(argc, argv))) {
        return 1;
    }

    if (unlikely(libdft_init() != 0)) {
        libdft_die();
        return 1;
    }

    syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
    syscall_set_pre(&syscall_desc[__NR_execve], pre_execve_hook);

    PIN_StartProgram();

    return 0;
}
```

main function

```
int main(int argc, char **argv) {
    PIN_InitSymbols(); // in case symbols are available

    if (unlikely(PIN_Init(argc, argv))) {
        return 1;
    }

    if (unlikely(libdft_init() != 0)) { // init data structures such as tagmap
        libdft_die(); // deallocate any resources libdft may have allocated
        return 1;
    }
    ...
}
```

- `unlikely(hoge)` tells compiler that `hoge` is unlikely to be true (= likely to be false).
 - Better branch prediction and less cycles.


main function

```
...
// socketcall events as the taint sources
syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
// taint sink
syscall_set_pre(&syscall_desc[__NR_execve], pre_execve_hook);

PIN_StartProgram(); // never returns

return 0;           // never reached
} // end of main
```

- `socketcall` events include `recv` and `recvfrom` events

 On some architectures—for example, x86-64 and ARM—there is no `socketcall()` system call; instead `socket(2)`, `accept(2)`, `bind(2)`, and so on really are implemented as separate system calls. (from `man socketcall`)

Details of func - alert

```
void alert(uintptr_t addr, const char *source, uint8_t tag) {  
    fprintf(stderr,  
        "\n(dta-execve) !!!!!!! ADDRESS 0x%x IS TAINTED (%s, tag=0x%02x), "  
        "ABORTING !!!!!!!\n",  
        addr, source, tag);  
    exit(1);  
}
```

- This alert function
 - i. prints an alert message with the details about the tainted address.
 - ii. exit from the application

Details of func - `check_string_taint`

```
void check_string_taint(const char *str, const char *source) {
    uint8_t tag;                                // to store "color"
    uintptr_t start = (uintptr_t)str;           // start of string
    uintptr_t end = (uintptr_t)str + strlen(str); // end of string

    fprintf(stderr, "(dta-execve) checking taint on bytes 0x%x -- 0x%x (%s)... ",
            start, end, source);

    for (uintptr_t addr = start; addr <= end; addr++) {
        tag = tagmap_getb(addr);                // get the "color" of addr
        if (tag != 0)
            alert(addr, source, tag);           // alert if tag is tainted
    }

    fprintf(stderr, "OK\n");
} // end of check_string_taint
```

Details of func - post_socketcall_hook

```
static void post_socketcall_hook(syscall_ctx_t *ctx) {
    int fd;
    void *buf;
    size_t len;

    int call = (int)ctx->arg[SYSCALL_ARG0];
    unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];

    switch (call) {
    case SYS_RECV:
    case SYS_RECVFROM:
        // ...
        // omitted
        // ...
        break;

    default:
        break;
    }
}
```


Details of func - `post_socketcall_hook`

```
static void post_socketcall_hook(syscall_ctx_t *ctx) {  
    int fd;           // these are used in the omitted part  
    void *buf;  
    size_t len;  
  
    int call = (int)ctx->arg[SYSCALL_ARG0]; // syscall number  
    unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];  
    ...  
}
```

- `ctx` : contains
 - the arguments that were passed to the syscall
 - the return value of syscall.

Details of func - `post_socketcall_hook`

```
...
switch (call) {
case SYS_RECV:
case SYS_RECVFROM:
    // ...
    // omitted
    // ...
    break;

default:
    break;
}
} // end of post_socketcall_hook
```

- Ignores any cases other than `SYS_RECV` or `SYS_RECVFROM`.
 - Thus, catching all the `socketcall` does work.

Details of func - post_socketcall_hook (omitted part)

```
... // start of omitted part
if (unlikely(ctx->ret <= 0)) {
    return;
}

fd = (int)args[0];
buf = (void *)args[1];
len = (size_t)ctx->ret;

fprintf(stderr, "(dta-execve) recv: %zu bytes from fd %u\n", len, fd);

for (size_t i = 0; i < len; i++) {
    if (isprint(((char *)buf)[i]))
        fprintf(stderr, "%c", ((char *)buf)[i]);
    else
        fprintf(stderr, "\\x%02x", ((char *)buf)[i]);
}
fprintf(stderr, "\n");

fprintf(stderr, "(dta-execve) tainting bytes %p -- 0x%x with tag 0x%x\n",
        buf, (uintptr_t)buf + len, 0x01);

tagmap_setn((uintptr_t)buf, len, 0x01);
... // end of omitted part
```

Details of func - `post_socketcall_hook` (omitted part)

```
... // start of omitted part
    if (unlikely(ctx->ret <= 0)) { // if syscall didn't receive any bytes
        return;
    }

    fd = (int)args[0];           // socket fd
    buf = (void *)args[1];       // buffer address
    len = (size_t)ctx->ret;       // # of received bytes

    fprintf(stderr, "(dta-execve) recv: %zu bytes from fd %u\n", len, fd);
...

```

Details of func - `post_socketcall_hook` (omitted part)

```
...
for (size_t i = 0; i < len; i++) { // print each char
    if (isprint(((char *)buf)[i]))
        fprintf(stderr, "%c", ((char *)buf)[i]);
    else
        fprintf(stderr, "\\x%02x", ((char *)buf)[i]);
}
fprintf(stderr, "\\n");

fprintf(stderr, "(dta-execve) tainting bytes %p -- 0x%x with tag 0x%x\\n",
        buf, (uintptr_t)buf + len, 0x01);

tagmap_setn((uintptr_t)buf, len, 0x01);
... // end of omitted part
```

- `isprint(hoge)` returns
 - none-0 if hoge can be printed
 - 0 if hoge can't be printed
- `buf` : first address that will be tainted
- `len` : # of bytes to taint
- `0x01` : taint color

Details of func - `post_socketcall_hook` (repeated)

```
static void post_socketcall_hook(syscall_ctx_t *ctx) {
    int fd;
    void *buf;
    size_t len;

    int call = (int)ctx->arg[SYSCALL_ARG0];
    unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];

    switch (call) {
    case SYS_RECV:
    case SYS_RECVFROM:
        // ...
        // omitted, just explained
        // ...
        break;

    default:
        break;
    }
}
```

Details of func - pre_execve_hook

```
static void pre_execve_hook(syscall_ctx_t *ctx) {
    const char *filename = (const char *)ctx->arg[SYSCALL_ARG0];
    char *const *args = (char *const *)ctx->arg[SYSCALL_ARG1];
    char *const *envp = (char *const *)ctx->arg[SYSCALL_ARG2];

    fprintf(stderr, "(dta-execve) execve: %s (@%p)\n", filename, filename);

    check_string_taint(filename, "execve command");
    while (args && *args) {
        fprintf(stderr, "(dta-execve) arg: %s (@%p)\n", *args, *args);
        check_string_taint(*args, "execve argument");
        args++;
    }
    while (envp && *envp) {
        fprintf(stderr, "(dta-execve) env: %s (@%p)\n", *envp, *envp);
        check_string_taint(*envp, "execve environment parameter");
        envp++;
    }
}
```

Details of func - `pre_execve_hook`

```
static void pre_execve_hook(syscall_ctx_t *ctx) {  
    const char *filename = (const char *)ctx->arg[SYSCALL_ARG0];  
    char *const *args = (char *const *)ctx->arg[SYSCALL_ARG1];  
    char *const *envp = (char *const *)ctx->arg[SYSCALL_ARG2];  
  
    fprintf(stderr, "(dta-execve) execve: %s (@%p)\n", filename, filename);  
    check_string_taint(filename, "execve command");  
    ...  
}
```

- taint sink
- `ctx` : contains
 - the arguments that were passed to the syscall
 - the return value of syscall.
- Check whether `filename` is tainted or not.

Details of func - `pre_execve_hook`

```
...  
while (args && *args) {  
    fprintf(stderr, "(dta-execve) arg: %s (@%p)\n", *args, *args);  
    check_string_taint(*args, "execve argument");  
    args++;  
}  
while (envp && *envp) {  
    fprintf(stderr, "(dta-execve) env: %s (@%p)\n", *envp, *envp);  
    check_string_taint(*envp, "execve environment parameter");  
    envp++;  
}  
}
```

- Check whether `args` and `envp` are tainted or not.

Test of Control-Flow Hijacking - `main`

```
int main(int argc, char *argv[]) {
    char buf[4096];
    struct sockaddr_storage addr;

    int sockfd = open_socket("localhost", "9999");

    socklen_t addrlen = sizeof(addr);
    recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);

    int child_fd = exec_cmd(buf);
    FILE *fp = fdopen(child_fd, "r");

    while (fgets(buf, sizeof(buf), fp)) {
        sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);
    }

    return 0;
}
```

- Some error-handling codes are omitted.

Test of Control-Flow Hijacking - `main`

```
int main(int argc, char *argv[]) {
    char buf[4096];
    struct sockaddr_storage addr;

    int sockfd = open_socket("localhost", "9999");

    socklen_t addrlen = sizeof(addr);
    recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);
    ...
}
```

1. Opens a socket.
2. Receives a message from the socket.

Test of Control-Flow Hijacking - `main`

```
...  
    int child_fd = exec_cmd(buf);  
    FILE *fp = fdopen(child_fd, "r");  
  
    while (fgets(buf, sizeof(buf), fp)) {  
        sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);  
    }  
  
    return 0;  
}
```

3. Executes a command.
 4. Writes the output of the command output to network socket.
- `exec_cmd` is vulnerable function.
 - The args of `execve` can be influenced by an attacker.

Test of Control-Flow Hijacking - cmd

```
static struct __attribute__((packed)) {  
    char prefix[32];  
    char datefmt[32];  
    char cmd[64];  
} cmd = {"date: ", "\\%Y-\\%m-\\%d \\%H:\\%M:\\%S",  
        "/home/binary/code/chapter11/date"};
```

- cmd contains a prefix for the command output from the message.

Test of Control-Flow Hijacking - `exec_cmd`

```
int exec_cmd(char *buf) {
    int pid; int p[2]; char *argv[3];

    for (i = 0; i < strlen(buf); i++) { // [1]
        if (buf[i] == '\n') {
            cmd.prefix[i] = '\0';
            break;
        }
        cmd.prefix[i] = buf[i]; // ** Buffer overflow **
    }

    argv[0] = cmd.cmd;
    argv[1] = cmd.datefmt;
    argv[2] = NULL;

    pipe(p) // omitted error-handling
    ...
}
```

- [1] lacks proper bound check, allowing attackers to overwrite the `cmd` field.

Test of Control-Flow Hijacking - `exec_cmd`

```
...
switch (pid = fork()) {
... // case -1:
case 0: /* Child */
    printf("(execve-test/child) execv: %s %s\n", argv[0], argv[1]);
    fflush(stdout);

    close(1);
    dup(p[1]);
    close(p[0]);

    printf("%s", cmd.prefix);
    fflush(stdout);
    execv(argv[0], argv);
... // error-handling
default: /* Parent */
    close(p[1]);
    return p[0];
}
return -1;
}
```

Test of Control-Flow Hijacking - `main` (repeated)

```
int main(int argc, char *argv[]) {
    char buf[4096];
    struct sockaddr_storage addr;

    int sockfd = open_socket("localhost", "9999");

    socklen_t addrlen = sizeof(addr);
    recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);

    int child_fd = exec_cmd(buf);
    FILE *fp = fdopen(child_fd, "r");

    while (fgets(buf, sizeof(buf), fp)) {
        sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);
    }

    return 0;
}
```


Test of Control-Flow Hijacking - test of overflow

No Buffer-Overflow

```
$ ./execve-test-overflow &  
[1] 1913  
$ nc -u 127.0.0.1 9999  
prefix!!!!:  
(execve-test/child) execv: /home/binary/code/chapter11/date %Y-%m-%d %H:%M:%S  
prefix!!!!: 2021-05-19 06:48:45  
^C[1]+ Done  
./execve-test-overflow
```

1. Start server. (localhost(=127.0.0.1), port 9999)
2. Send prefix to localhost:9999 using `nc`.



`-u` option : Use UDP instead of TCP.

Test of Control-Flow Hijacking - test of overflow

Buffer-Overflow

```
$ ./execve-test-overflow &
[1] 2061
$ uc -u 127.0.0.1 9999
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb/home/binary/code/chapter11/echo
(execve-test/child) execv: /home/binary/code/chapter11/echo bb...bb/home/binary/.../echo
aa...aabb...bb/home/binary/code/chapter11/echo bb...bb/home/binary/code/chapter11/echo
^C[1]+ Done
./execve-test-overflow
```

```
static struct __attribute__((packed)) {
    char prefix[32]; // aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    char datefmt[32]; // bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
    char cmd[64]; // /home/binary/code/chapter11/echo <- This command is executed.
} cmd;
```



`-u` option : Use UDP instead of TCP.

Test of Control-Flow Hijacking - Detect Hijacking

Using DTA to Detect Hijacking Attempt

```
$ cd /home/binary/lidft/pin-2.13-61206-gcc.4.4.7-linux/
$ ./pin.sh -follow_execv -t /home/binary/code/chapter11/dta-execve.so \
    -- /home/binary/code/chapter11/execve-test-overflow &
[1] 2994
$ nc -u 127.0.0.1 9999
aaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbb/home/binary/code/chapter11/echo
(dta-execve) recv: 97 bytes from fd 4
aa...abb...bb/home/binary/code/chapter11/echo\x0a
(dta-execve) taintin bytes 0xffff9499c -- 0xffff949fd with tag 0x1 # taints all the bytes received
(execve-test/child) execv: /home/binary/code/chapter11/echo bb...bb/home/binary/.../echo
(dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
(dta-execve) checking taint on bytes 0x804b100 -- 0x804b120 (execve command)...
(dta-execve) !!!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01), ABORTING !!!!!!!
```

1. Check whether arguments of `execve` are tainted.
2. `dta-execve` notices that the command is tainted with `0x01`.
3. Raises an alert and then stops the child process.

Overview

1. Internals of `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circumventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Circumventing DTA with Implicit Flows

```
int exec_cmd(char *buf) {  
    ...  
    for (i = 0; i < strlen(buf); i++) {  
        if (buf[i] == '\n') {  
            cmd.prefix[i] = '\0';  
            break;  
        }  
        c = 0;  
        while (c < buf[i]) c++; // increment c until c == b[i]  
        cmd.prefix[i] = c;      // c == b[i], but c is not tainted  
    }  
    ... // Set up argv and continue with execv  
}
```

- Without explicitly copying `b[i]` to `c`, `c` has the same value as `b[i]`.
- We call this *implicit flow*.
 - `libdft` cannot track this kind of flow, causing undertainting.
- Malware may contain implicit flow to confuse taint analysis.

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circumventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector
 - Header files
 - Data structure
 - Functions
 - `main` function
 - Details of func
 - Test of Data Exfiltration

A DTA-Based Data Exfiltration Detector

- We use multiple taint color so that we can tell *which* file is leaking.
 - In the previous example, single taint color is enough to detect bytes are attacker-controlled or not.
- Taint settings
 - Taint source : `open` and `read`
 - Taint sink : `socketcall`, such as `send`, `sendto`

Header files

```
#include "pin.H"

#include "branch_pred.h"
#include "libdft_api.h"
#include "syscall_desc.h"
#include "tagmap.h"
```

- all `libdft` tools are just Pin tools ilnked with the `libdft` library.
- This is same as the previous example.

Data Structure

```
extern syscall_desc_t syscall_desc[SYSCALL_MAX];    // to hook syscalls
static std::map<int, uint8_t> fd2color;
static std::map<uint8_t, std::string> color2fname;  // colors -> filenames

#define MAX_COLOR 0x80  // possible maximum color value
```

- `fd2color` : maps file descriptors to colors
- `color2fname` : maps taint colors to filenames

Functions

```
void alert(uintptr_t addr, uint8_t tag);  
static void post_open_hook(syscall_ctx_t *ctx);  
static void post_read_hook(syscall_ctx_t *ctx);  
static void pre_socketcall_hook(syscall_ctx_t *ctx);
```

- `post_open_hook` / `post_read_hook` runs after `open` / `read` syscall respectively.
- `pre_socketcall_hook` runs before the socketcall syscall such as `recv` or `recvfrom`.

main function

```
int main(int argc, char **argv) {
    PIN_InitSymbols();
    if (unlikely(PIN_Init(argc, argv))) {
        return 1;
    }
    if (unlikely(libdft_init() != 0)) {
        libdft_die();
        return 1;
    }

    syscall_set_post(&syscall_desc[__NR_open], post_open_hook);
    syscall_set_post(&syscall_desc[__NR_read], post_read_hook);
    syscall_set_pre(&syscall_desc[__NR_socketcall], pre_socketcall_hook);
    PIN_StartProgram();

    return 0;
}
```

- `main` func is almost identical to that of the previous example.

Details of func `alert`

```
void alert(uintptr_t addr, uint8_t tag) {
    fprintf(stderr,
        "\n(dta-dataleak) !!!!!!! ADDRESS 0x%x IS TAINTED (tag=0x%02x), "
        "ABORTING !!!!!!!\n",
        addr, tag);

    for (unsigned c = 0x01; c <= MAX_COLOR; c <= 1) {
        if (tag & c) {
            fprintf(stderr, "  tainted by color = 0x%02x (%s)\n", c,
                color2fname[c].c_str());
        }
    }
    exit(1);
}
```

- Alert which address is tainted and with which color.
- It's possible that the data is tainted with multiple color (= multiple files).

Details of func - `post_open_hook`

```
static void post_open_hook(syscall_ctx_t *ctx) {  
    static uint8_t next_color = 0x01;  
    uint8_t color;  
    int fd = (int)ctx->ret; // return value of syscall open  
    const char *fname = (const char *)ctx->arg[SYSCALL_ARG0]; // filename to open  
    ...  
}
```

- `ctx->ret` : contains return value of syscall.
 - In this case, return value is the file descriptor that was opened.
- `fname` : filename that was opened.

Details of func - `post_open_hook`

```
static void post_open_hook(syscall_ctx_t *ctx) {  
    ...  
    if (unlikely((int)ctx->ret < 0)) {  
        return;  
    }  
    if (strstr(fname, ".so") || strstr(fname, ".so.")) {  
        return;  
    }  
    ...  
}
```

1. Checks whether the return value is not smaller than 0.
 - You don't need to taint if `open` is failed.
2. Filters out uninteresting files such as shared libraries.
 - Shared libraries don't have any secret informations.
 - In a real-world DTA tool, you should filter out some more files.

Details of func - `post_open_hook`

```
static void post_open_hook(syscall_ctx_t *ctx) {  
    ...  
    if (!fd2color[fd]) {  
        color = next_color;  
        fd2color[fd] = color;  
        if (next_color < MAX_COLOR) next_color <= 1; // static variable  
    } else {  
        color = fd2color[fd]; // reuse color of a file with the same fd  
    }  
    ...  
}
```

- "color" can be reused
 - if a file descriptor is closed and then the same file descriptor is reused.
- `MAX_COLOR` can be assigned to many `fd`
 - if we run out of "color".
 - We only supports 8 colors (because `color` is 8-bit wise).

Details of func - `post_open_hook`

```
static void post_open_hook(syscall_ctx_t *ctx) {  
    ...  
    if (color2fname[color].empty())  
        color2fname[color] = std::string(fname);  
    else  
        color2fname[color] += " | " + std::string(fname);  
}
```

- Update the `color2fname` map with just opened filename.
- Filename is concatenated with " | "
 - if taint color is reused for multiple files.

Details of func - `post_read_hook`

```
static void post_read_hook(syscall_ctx_t *ctx) {  
    int fd      = (int)ctx->arg[SYSCALL_ARG0];  
    void *buf    = (void*)ctx->arg[SYSCALL_ARG1];  
    size_t len   = (size_t)ctx->ret;  
    uint8_t color;  
    ...  
}
```

- `fd` : file descriptor that's being read
- `buf` : buffer into which bytes are read
- `len` : length of buffer that was read.

Details of func - `post_read_hook`

```
static void post_read_hook(syscall_ctx_t *ctx) {  
    ...  
    if(unlikely(len <= 0)) {  
        return;  
    }  
  
    fprintf(stderr, "(dta-dataleak) read: %zu bytes from fd %u\n", len, fd);  
    ...  
}
```

- You don't need to taint if nothing was read, that is when...
 - i. 0 byte was read(, when return value is 0).
 - ii. `read` failed (, when return value is negative).

Details of func - `post_read_hook`

```
static void post_read_hook(syscall_ctx_t *ctx) {  
    ...  
    color = fd2color[fd];  
    if(color) {  
        tagmap_setn((uintptr_t)buf, len, color);  
    } else {  
        tagmap_clrn((uintptr_t)buf, len);  
    }  
}
```

- Taint bytes using `tagmap_setn()`
 - if the `fd` is colored.
- Clear taint on bytes using `tagmap_clrn()`
 - if the `fd` is not colored.

Details of func - pre_socketcall_hook

```
static void pre_socketcall_hook(syscall_ctx_t *ctx) {
    int fd;
    void *buf;
    size_t i, len;
    uint8_t tag;
    uintptr_t start, end, addr;

    int call = (int)ctx->arg[SYSCALL_ARG0];
    unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];

    switch(call) {
    case SYS_SEND:
    case SYS_SENDDTO:
        ...
        break;

    default:
        break;
    }
}
```

Details of func - pre_socketcall_hook

```
static void pre_socketcall_hook(syscall_ctx_t *ctx) {  
    ...  
    int call = (int)ctx->arg[SYSCALL_ARG0];  
    unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];  
    ...  
}
```

- `call` : type(number) of socketcall
 - such as `recv`, `recvfrom`, `send`, `sendto`.
- `args` : arguments that was passed to socketcall

Details of func - pre_socketcall_hook

```
static void pre_socketcall_hook(syscall_ctx_t *ctx) {  
    ...  
    switch(call) {  
    case SYS_SEND:  
    case SYS_SENDDTO:  
        ... // omitted  
        break;  
  
    default:  
        break;  
    }  
}
```

- Check the tagmap, if the socketcall is `send` or `sendto`.

Details of func - `pre_socketcall_hook` (omitted part)

```
static void pre_socketcall_hook(syscall_ctx_t *ctx) {  
    ...  
    fd = (int)args[0];  
    buf = (void*)args[1];  
    len = (size_t)args[2];  
  
    start = (uintptr_t)buf;  
    end = (uintptr_t)buf+len;  
    for(addr = start; addr <= end; addr++) {  
        tag = tagmap_getb(addr);  
        if(tag != 0) alert(addr, tag);  
    }  
    ...  
}  
}
```

- Loops over all of bytes in the send buffer and check whether they are tainted or not.
- If tainted, alert and exit the application.

Test of Data Exfiltration - dataleak-test-xor - main

```
int main(int argc, char *argv[]) {
    size_t i, j, k;
    FILE *fp[10];
    char buf1[4096], buf2[4096], *filenames[10];
    struct sockaddr_storage addr;

    srand(time(NULL)); // set seed

    int sockfd = open_socket("localhost", "9999"); // 1

    socklen_t addrlen = sizeof(addr);
    recvfrom(sockfd, buf1, sizeof(buf1), 0, (struct sockaddr*)&addr, &addrlen); // 2
    ...
}
```

1. Open a socket.
2. Read filenames from the socket.

Test of Data Exfiltration - dataleak-test-xor - main

```
int main(int argc, char *argv[]) {  
    ...  
    size_t fcount = split_filenames(buf1, filenames, 10); // 1  
    for(i = 0; i < fcount; i++) {  
        fp[i] = fopen(filenames[i], "r"); // 2  
    }  
    i = rand() % fcount; // 3  
    do { j = rand() % fcount; } while(j == i);  
  
    memset(buf1, '\\0', sizeof(buf1)); // initialize buffer  
    memset(buf2, '\\0', sizeof(buf2));  
    ...  
}
```

1. Gets each filenames.
2. Opens all the requested files.
3. Choses two of the opened files at random.

Test of Data Exfiltration - dataleak-test-xor - main

```
int main(int argc, char *argv[]) {  
    ...  
    while(fgets(buf1, sizeof(buf1), fp[i]) && fgets(buf2, sizeof(buf2), fp[j])) {  
        for(k = 0; k < sizeof(buf1)-1 && k < sizeof(buf2)-1; k++) {  
            buf1[k] ^= buf2[k];  
        }  
        sendto(sockfd, buf1, strlen(buf1)+1, 0, (struct sockaddr*)&addr, addrlen);  
    }  
    return 0;  
}
```

- Reads each files line by line, concatenating each pair of lines by operating XOR and sending to the socket.
- `buf[sizeof(buf) - 1]` would be a NULL character, so you should loop over from `0` to `sizeof(buf) - 2`.

Test of Data Exfiltration

```
$ ./pin.sh -follow-execv -t ~/code/chapter11/dta-dataleak.so -- ~/code/chapter11/dataleak-test-xor &
(dta-dataleak) read: 512 bytes from fd 4
(dta-dataleak) clearing taint on bytes 0xffb4aa80 -- 0xffb4ac80

$ nc -u 127.0.0.1 9999
/home/binary/code/chapter11/dta-execve.cpp ../dta-dataleak.cpp ../date.c ../echo.c
```

1. Runs `dataleak-test-xor` server with `dta-dataleak` as Pin tool,
 - immediately loading `dataleak-test-xor` itself.
2. Starts `netcat` session to connect to the server.
3. Sends a list of filenames to open.

Test of Data Exfiltration

```
$ nc -u 127.0.0.1 9999
/home/binary/code/chapter11/dta-execve.cpp .../dta-dataleak.cpp .../date.c .../echo.c
(dta-dataleak) opening /home/binary/code/chapter11/dta-execve.cpp at fd 5 with color 0x01
(dta-dataleak) opening /home/binary/code/chapter11/dta-dataleak.cpp at fd 6 with color 0x02
(dta-dataleak) opening /home/binary/code/chapter11/date.c at fd 7 with color 0x04
(dta-dataleak) opening /home/binary/code/chapter11/echo.c at fd 8 with color 0x08
...
```

1. Assigns each of files with a taint color.

Test of Data Exfiltration

```
$ nc -u 127.0.0.1 9999
/home/binary/code/chapter11/dta-execve.cpp .../dta-dataleak.cpp .../date.c .../echo.c
...
(dta-dataleak) read: 4096 bytes from fd 6
(dta-dataleak) tainting bytes 0x9b775c0 -- 0x9b785c0 with color 0x2
(dta-dataleak) read: 155 bytes from fd 8
(dta-dataleak) tainting bytes 0x9b785c8 -- 0x9b67663 with color 0x8
(dta-dataleak) send: 20 bytes from fd 4
...
```

1. Randomly chooses two files to leak,
 - that is, 6 and 8.
2. Intercepts the server's attempt to send the contents of files.

Test of Data Exfiltration

```
$ nc -u 127.0.0.1 9999
/home/binary/code/chapter11/dta-execve.cpp ../dta-dataleak.cpp ../date.c ../echo.c
...
(dta-dataleak) checking taint on bytes 0xffb48f7c -- 0xffb48f90...
(dta-dataleak) !!!!!!! ADDRESS 0xffb48f7c IS TANTED (tag=0x0a), ABORTING !!!!!!!
    tainted by color = 0x02 (/home/binary/code/chapter11/dta-dataleak.cpp)
    tainted by color = 0x08 (/home/binary/code/chapter11/echo.c)
^C
```

1. Checks the taint color of the contents,
 - detecting that they're tainted.