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. Circumeventing 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. Circumeventing 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
  - althogh 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.(OALD)
- `libdft` is based on Pin (between 2.11 and 2.13)
- 64-bit version of `libdft` : https://github.com/AngoraFuzzer/libdft64

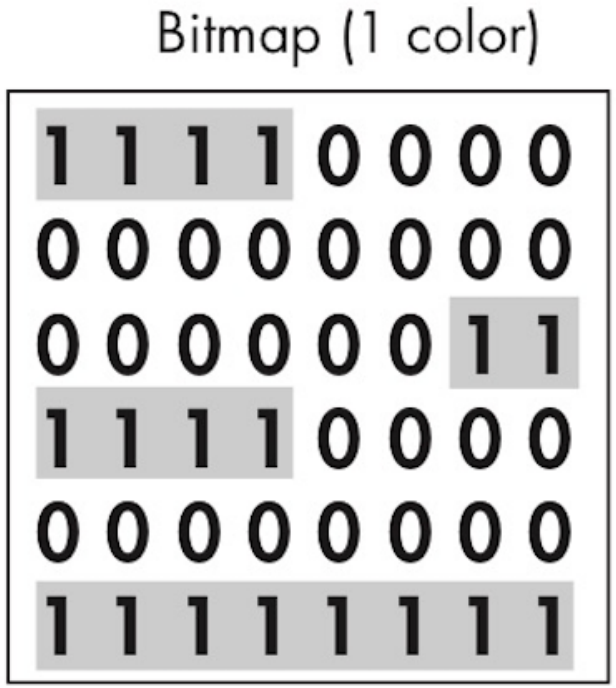
Overview

1. Internals of `libdft`
    - Shadow Memory
        - how to store taint info
    - Virtual CPU
        - how to propagete taint info
    - The libdft API and I/O interface
        - how to instrument
    - Taint Policy
2. Using DTA to Detect Remote Control-Hijacking
3. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Internals of libdft - Shadow Memory

Bitmap (1 color)

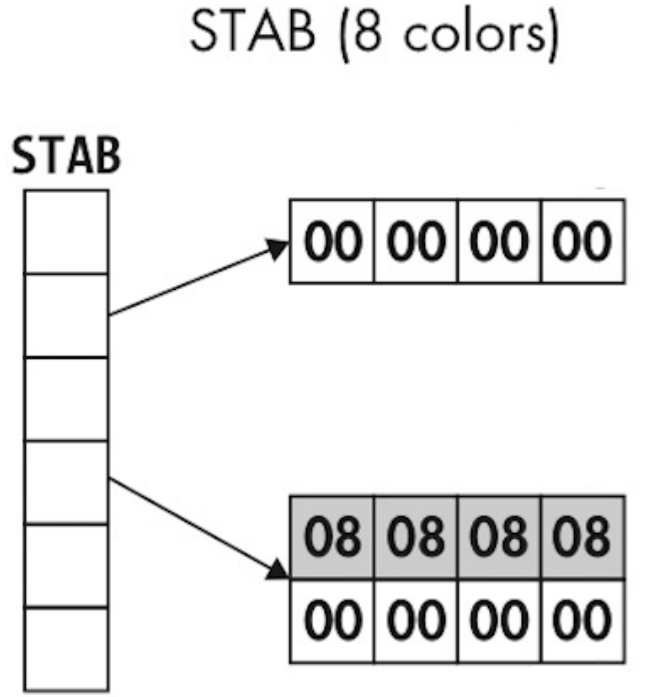
- Supports only 1 taint color.
- Slightly faster and use less memory than STAB.

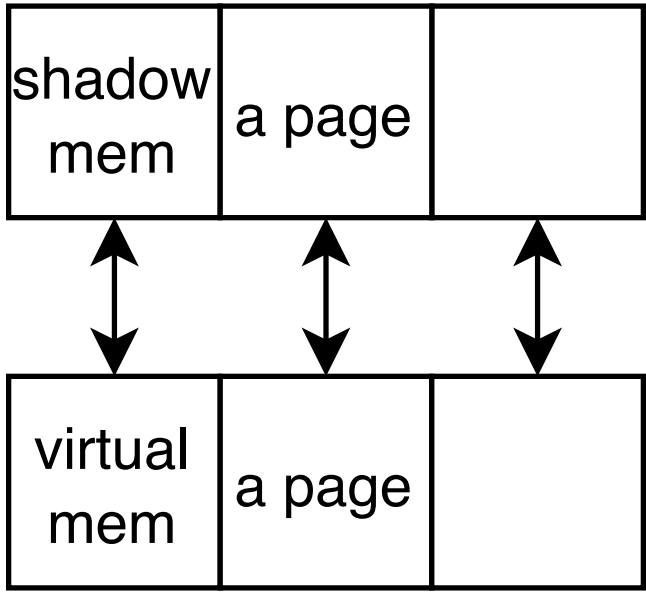


Internals of libdft - Shadow Memory

STAB : Segment Translation Table (8 color)

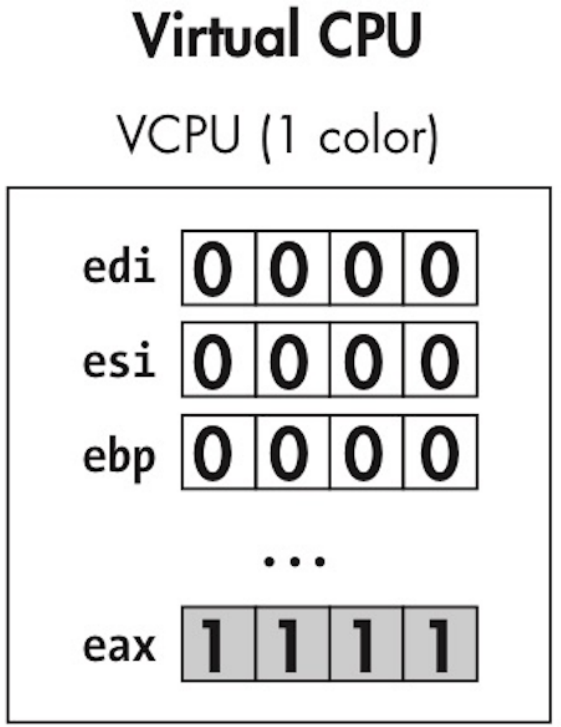
- Supports 8 taint color (because we use 8 bits to contain color).
- Contains one entry for every memory page.
  - and is allocatd in page-sized chunks
- Input and Output of STAB
  - input : some upper bits of virtual memory address
    - 16 bit if each page size is 2^{16}B(= 64KB)
  - output : some 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 virutual memory is adjacent.

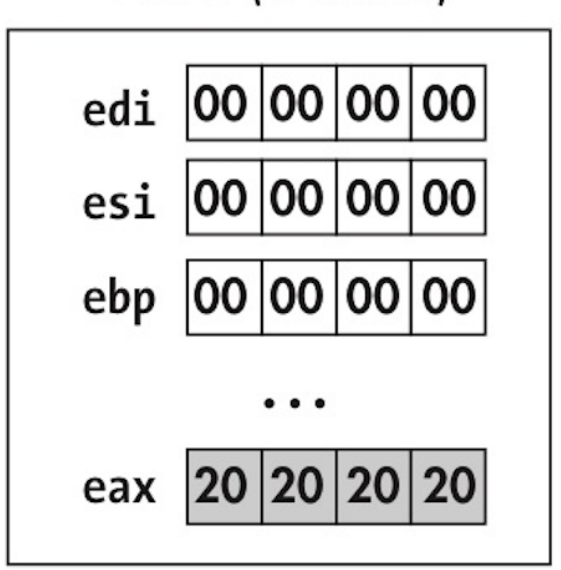




Internals of libdft - Virtual CPU

- *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`

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 shadow memory (Tagmap API)
  - add callbacks and instrument code

tagmap : shadow memory

Internals of libdft - libdft API and I/O interface

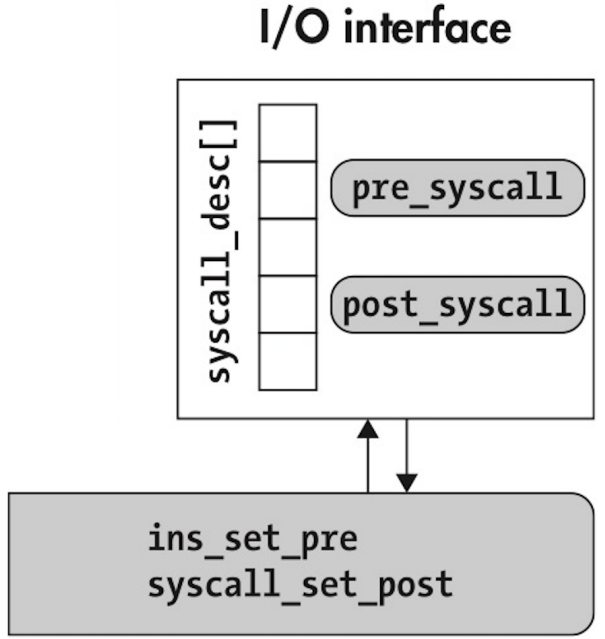
- `libdft` provides a taint tracking API.
- Two import tools for building DTA tools is those that
  - manipulate tagmap (Tagmap API) ←
  - add callbacks and instrument code

Tagmap API

- `tagmap_setb()` : sets status of tagmap in byte granularity.
  - `tagmap_setn()` : taints arbitary number of bytes.
- `tagmap_getb()` : gets status of tagmap in byte granularity.
  - `tagmap_getn()` : checks arbitary 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 classess define how to propagete 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 opperands
4. SPECIAL : instructions that require special rules
5. FPU, MMX, SSE : instructions `libdft` doesn't supports. Doesn't propagete 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. Circumeventing 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 (-> arbitary code execution).
- Taint settings
  - taint source : the network receive functions, `recv` and `recvfrom`.
  - tiant sink : `execve`

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Examples of `exec` family

	引数の渡し方(l or v)	環境変数(e)	パス検索(p)
execl	list	引き継ぐ	しない
execv	vector	引き継ぐ	しない
execle	list	渡す	しない
execve	vector	渡す	しない
execlp	list	引き継ぐ	する
execvp	vector	引き継ぐ	する

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

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Header files

```
01 // dta-execve.cpp
02 ...
03 #include "pin.H"              // libdft uses Pin
04
05 #include "branch_pred.h"     // hints for branch prediction
06 #include "libdft_api.h"      // lidft base API
07 #include "syscall_desc.h"
08 #include "tagmap.h"
09 ...
```

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

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Functions

```
01 extern syscall_desc_t syscall_desc[SYSCALL_MAX];          // store syscall hooks
02
03 void alert(uintptr_t addr, const char *source, uint8_t tag);   // alert and exit
04 void check_string_taint(const char *str, const char *source);  // check color
05 static post_socketcall_hook(syscall_ctx_t *ctx);               // taint source
06 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.

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

`main` function

```
01 int main(int argc, char **argv) {
02   PIN_InitSymbols();
03
04   if (unlikely(PIN_Init(argc, argv))) {
05     return 1;
06   }
07
08   if (unlikely(libdft_init() != 0)) {
09     libdft_die();
10     return 1;
11   }
12
13   syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
14   syscall_set_pre(&syscall_desc[__NR_execve], pre_execve_hook);
15
16   PIN_StartProgram();
17
18   return 0;
19 }
```

`main` function

```
01 int main(int argc, char **argv) {
02   PIN_InitSymbols();  // in case symbols are available
03
04   if (unlikely(PIN_Init(argc, argv))) {
05     return 1;
06   }
07
08   if (unlikely(libdft_init() != 0)) {  // init data structures such as tagmap
09     libdft_die();  // deallocat any resources libdft may have allocated
10     return 1;
11   }
12 ...
```

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

`main` function

```
01 ...
02   // socketcall events as the taint sources
03   syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
04   // taint sink
05   syscall_set_pre(&syscall_desc[__NR_execve], pre_execve_hook);
06
07   PIN_StartProgram();  // never returns
08
09   return 0;            // never reached
10 }  // 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`)

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Details of func - `alert`

```
01 void alert(uintptr_t addr, const char *source, uint8_t tag) {
02   fprintf(stderr,
03           "\n(dta-execve) !!!!!!! ADDRESS 0x%x IS TAINTED (%s, tag=0x%02x), "
04           "ABORTING !!!!!!!\n",
05           addr, source, tag);
06   exit(1);
07 }
```

- 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`

```
01 void check_string_taint(const char *str, const char *source) {
02   uint8_t tag;                                    // to store "color"
03   uintptr_t start = (uintptr_t)str;               // start of string
04   uintptr_t end = (uintptr_t)str + strlen(str);   // end of string
05
06   for (uintptr_t addr = start; addr <= end; addr++) {
07     tag = tagmap_getb(addr);                      // get the "color" of addr
08     if (tag != 0)
09       alert(addr, source, tag);                   // alert if tag is tainted
10   }
11
12   fprintf(stderr, "OK\n");
13 }  // end of check_string_taint
```

Details of func - `post_socketcall_hook`

```
01 static void post_socketcall_hook(syscall_ctx_t *ctx) {
02   int fd;
03   void *buf;
04   size_t len;
05
06   int call = (int)ctx->arg[SYSCALL_ARG0];
07   unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];
08
09   switch (call) {
10   case SYS_RECV:
11   case SYS_RECVFROM:
12     // ...
13     // omitted
14     // ...
15     break;
16
17   default:
18     break;
19   }
20 }
```

Details of func - `post_socketcall_hook`

```
01 static void post_socketcall_hook(syscall_ctx_t *ctx) {
02   int fd;       // these are used in the omitted part
03   void *buf;
04   size_t len;
05
06   int call = (int)ctx->arg[SYSCALL_ARG0];  // syscall number
07   unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];
08 ...
```

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

Details of func - `post_socketcall_hook`

```
01 ...
02   switch (call) {
03   case SYS_RECV:
04   case SYS_RECVFROM:
05     // ...
06     // omitted
07     // ...
08     break;
09
10   default:
11     break;
12   }
13 }  // 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)

```
01 ... // start of omitted part
02     if (unlikely(ctx->ret <= 0)) {
03       return;
04     }
05
06     fd = (int)args[0];
07     buf = (void *)args[1];
08     len = (size_t)ctx->ret;
09
10     for (size_t i = 0; i < len; i++) {
11       if (isprint(((char *)buf)[i]))
12         fprintf(stderr, "%c", ((char *)buf)[i]);
13       else
14         fprintf(stderr, "\\x%02x", ((char *)buf)[i]);
15     }
16     fprintf(stderr, "\n");
17
18     tagmap_setn((uintptr_t)buf, len, 0x01);
19 ... // end of omitted part
```

Details of func - `post_socketcall_hook` (omitted part)

```
01 ... // start of omitted part
02     if (unlikely(ctx->ret <= 0)) {  // if syscall didn't receive any bytes
03       return;
04     }
05
06     fd = (int)args[0];              // socket fd
07     buf = (void *)args[1];          // buffer address
08     len = (size_t)ctx->ret;         // # of received bytes
09 ...
```

Details of func - `post_socketcall_hook` (omitted part)

```
01 ...
02     for (size_t i = 0; i < len; i++) {  // print each char
03       if (isprint(((char *)buf)[i]))
04         fprintf(stderr, "%c", ((char *)buf)[i]);
05       else
06         fprintf(stderr, "\\x%02x", ((char *)buf)[i]);
07     }
08     fprintf(stderr, "\n");
09
10     tagmap_setn((uintptr_t)buf, len, 0x01);
11 ... // end of omited 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)

```
01 static void post_socketcall_hook(syscall_ctx_t *ctx) {
02   int fd;
03   void *buf;
04   size_t len;
05
06   int call = (int)ctx->arg[SYSCALL_ARG0];
07   unsigned long *args = (unsigned long *)ctx->arg[SYSCALL_ARG1];
08
09   switch (call) {
10   case SYS_RECV:
11   case SYS_RECVFROM:
12     // ...
13     // omitted, just explained
14     // ...
15     break;
16
17   default:
18     break;
19   }
20 }
```

Details of func - `pre_execve_hook`

```
01 static void pre_execve_hook(syscall_ctx_t *ctx) {
02   const char *filename = (const char *)ctx->arg[SYSCALL_ARG0];
03   char *const *args = (char *const *)ctx->arg[SYSCALL_ARG1];
04   char *const *envp = (char *const *)ctx->arg[SYSCALL_ARG2];
05
06   check_string_taint(filename, "execve command");
07   while (args && *args) {
08     fprintf(stderr, "(dta-execve) arg: %s (@%p)\n", *args, *args);
09     check_string_taint(*args, "execve argument");
10     args++;
11   }
12   while (envp && *envp) {
13     fprintf(stderr, "(dta-execve) env: %s (@%p)\n", *envp, *envp);
14     check_string_taint(*envp, "execve environment parameter");
15     envp++;
16   }
17 }
```

Details of func - `pre_execve_hook`

```
01 static void pre_execve_hook(syscall_ctx_t *ctx) {
02   const char *filename = (const char *)ctx->arg[SYSCALL_ARG0];
03   char *const *args = (char *const *)ctx->arg[SYSCALL_ARG1];
04   char *const *envp = (char *const *)ctx->arg[SYSCALL_ARG2];
05
06   check_string_taint(filename, "execve command");
07 ...
```

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

Details of func - `pre_execve_hook`

```
01 ...
02   while (args && *args) {
03     fprintf(stderr, "(dta-execve) arg: %s (@%p)\n", *args, *args);
04     check_string_taint(*args, "execve argument");
05     args++;
06   }
07   while (envp && *envp) {
08     fprintf(stderr, "(dta-execve) env: %s (@%p)\n", *envp, *envp);
09     check_string_taint(*envp, "execve environment parameter");
10     envp++;
11   }
12 }
```

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

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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Test of Control-Flow Hijacking - `main`

```
01 int main(int argc, char *argv[]) {
02   char buf[4096];
03   struct sockaddr_storage addr;
04
05   int sockfd = open_socket("localhost", "9999");
06
07   socklen_t addrlen = sizeof(addr);
08   recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);
09
10   int child_fd = exec_cmd(buf);
11   FILE *fp = fdopen(child_fd, "r");
12
13   while (fgets(buf, sizeof(buf), fp)) {
14     sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);
15   }
16
17   return 0;
18 }
```

- Some error-handling codes are omitted.

Test of Control-Flow Hijacking - `main`

```
01 int main(int argc, char *argv[]) {
02   char buf[4096];
03   struct sockaddr_storage addr;
04
05   int sockfd = open_socket("localhost", "9999");  // 1
06
07   socklen_t addrlen = sizeof(addr);
08   // 2
09   recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);
10 ...
```

1. Open a socket.
2. Receives a meassage from the socket.

Test of Control-Flow Hijacking - `main`

```
01 ...
02   int child_fd = exec_cmd(buf);  // 3
03   FILE *fp = fdopen(child_fd, "r");
04
05   while (fgets(buf, sizeof(buf), fp)) {
06     // 4
07     sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);
08   }
09
10   return 0;
11 }
```

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`

```
01 static struct __attribute__((packed)) {
02   char prefix[32];  // <- read from the Network
03   char datefmt[32];
04   char cmd[64];
05 } cmd = {"date: ", "\%Y-\%m-\%d \%H:\%M:\%S",
06          "/home/binary/code/chapter11/date"};
```

- `cmd` contains:
  - `prefix` for the command output
  - `datefmt` for the output of `date` command
  - `cmd`, `date` itself

Test of Control-Flow Hijacking - `exec_cmd`

```
01 int exec_cmd(char *buf) {
02   int pid; int p[2]; char *argv[3];
03
04   for (i = 0; i < strlen(buf); i++) {  // [1]
05     if (buf[i] == '\n') {
06       cmd.prefix[i] = '\0';
07       break;
08     }
09     cmd.prefix[i] = buf[i];  // ** Buffer overflow **
10   }
11
12   argv[0] = cmd.cmd;
13   argv[1] = cmd.datefmt;
14   argv[2] = NULL;
15
16   pipe(p)  // omitted error-handling
17 ...
```

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

Test of Control-Flow Hijacking - `exec_cmd`

```
01 ...
02   switch (pid = fork()) {
03   ...  // case -1:
04   case 0: /* Child */
05     printf("(execve-test/child) execv: %s %s\n", argv[0], argv[1]);
06     fflush(stdout);
07
08     close(1);
09     dup(p[1]);
10     close(p[0]);
11
12     printf("%s", cmd.prefix);
13     fflush(stdout);
14     execv(argv[0], argv);
15     ...  // error-handling
16   default: /* Parent */
17     close(p[1]);
18     return p[0];
19   }
20   return -1;
   }
```

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

```
01 int main(int argc, char *argv[]) {
02   char buf[4096];
03   struct sockaddr_storage addr;
04
05   int sockfd = open_socket("localhost", "9999");
06
07   socklen_t addrlen = sizeof(addr);
08   recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &addrlen);
09
10   int child_fd = exec_cmd(buf);
11   FILE *fp = fdopen(child_fd, "r");
12
13   while (fgets(buf, sizeof(buf), fp)) {
14     sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&addr, addrlen);
15   }
16
17   return 0;
18 }
```

Test of Control-Flow Hijacking - test of overflow

No Buffer-Overflow

```
01 $ ./execve-test-overflow &
02 [1] 1913
03 $ nc -u 127.0.0.1 9999
04 prefix!!!!:
05 (execve-test/child) execv: /home/binary/code/chapter11/date %Y-%m-%d %H:%M:%S
06 prefix!!!!: 2021-05-19 06:48:45
07 ^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
$ nc -u 127.0.0.1 9999
aa..aabb..bb/home/binary/code/chapter11/echo
(execve-test/child) execv: /.../code/chapter11/echo bb...bb/home/binary/.../echo
aa...aabb...bb/home/binary/code/chapter11/echo bb...bb/home/.../chapter11/echo
^C[1]+ Done                          ./execve-test-overflow
```

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

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

Test of Control-Flow Hijacking - Detect Hijacking

Using DTA to Detect Hijacking Attempt

```
01 $ cd /home/binary/lidft/pin-2.13-61206-gcc.4.4.7-linux/
02 $ ./pin.sh -follow_execv -t /home/binary/code/chapter11/dta-execve.so \
03          -- /home/binary/code/chapter11/execve-test-overflow &
04 $ nc -u 127.0.0.1 9999
05 aa..aabb..bb/home/binary/code/chapter11/echo
06 (dta-execve) recv: 97 bytes from fd 4
07 aa...aabb...bb/home/binary/code/chapter11/echo\x0a
08 (dta-execve) tainting bytes 0xfff9499c -- 0xfff949fd with tag 0x1
09 (execve-test/child) execv: /home/binary/../echo bb...bb/home/binary/.../echo
10 (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
11 (dta-execve) checking taint on bytes 0x804b100 -- 0x804b120 (execve command)...
12 (dta-execve)
13 !!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01), ABORTING !!!!!
```

1. Check whether argments 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. Circumeventing DTA with implicit Flows
4. A DTA-Based Data Exfiltration Detector

Circumeventing DTA with Implicit Flows

```
01 int exec_cmd(char *buf) {
02 ...
03   for (i = 0; i < strlen(buf); i++) {
04     if (buf[i] == '\n') {
05       cmd.prefix[i] = '\0';
06       break;
07     }
08     c = 0;
09     while (c < buf[i]) c++;  // increment c until c == b[i]
10     cmd.prefix[i] = c;       // c == b[i], but c is not tainted
11   }
12 ...  // Set up argv and continue with execv
13 }
```

- 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. Circemeventing 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`

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

Header files

```
01 #include "pin.H"
02
03 #include "branch_pred.h"
04 #include "libdft_api.h"
05 #include "syscall_desc.h"
06 #include "tagmap.h"
```

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

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

Data Structure

```
01 extern syscall_desc_t syscall_desc[SYSCALL_MAX];    // to hook syscalls
02 static std::map<int, uint8_t> fd2color;
03 static std::map<uint8_t, std::string> color2fname;  // colors -> filenames
04
05 #define MAX_COLOR 0x80  // possible maximum color value
```

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

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

Functions

```
01 void alert(uintptr_t addr, uint8_t tag);
02 static void post_open_hook(syscall_ctx_t *ctx);
03 static void post_read_hook(syscall_ctx_t *ctx);
04 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`.

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

`main` function

```
01 int main(int argc, char **argv) {
02   PIN_InitSymbols();
03   if (unlikely(PIN_Init(argc, argv))) {
04     return 1;
05   }
06   if (unlikely(libdft_init() != 0)) {
07     libdft_die();
08     return 1;
09   }
10
11   syscall_set_post(&syscall_desc[__NR_open], post_open_hook);
12   syscall_set_post(&syscall_desc[__NR_read], post_read_hook);
13   syscall_set_pre(&syscall_desc[__NR_socketcall], pre_socketcall_hook);
14   PIN_StartProgram();
15
16   return 0;
17 }
```

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

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

Details of func `alert`

```
01 void alert(uintptr_t addr, uint8_t tag) {
02   fprintf(stderr,
03           "\n(dta-dataleak) !!!!!!! ADDRESS 0x%x IS TAINTED (tag=0x%02x), "
04           "ABORTING !!!!!!!\n",
05           addr, tag);
06
07   for (unsigned c = 0x01; c <= MAX_COLOR; c <<= 1) {
08     if (tag & c) {
09       fprintf(stderr, "  tainted by color = 0x%02x (%s)\n", c,
10               color2fname[c].c_str());
11     }
12   }
13   exit(1);
14 }
```

- 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`

```
01 static void post_open_hook(syscall_ctx_t *ctx) {
02   static uint8_t next_color = 0x01;
03   uint8_t color;
04   int fd = (int)ctx->ret;  // return value of syscall open
05   const char *fname = (const char *)ctx->arg[SYSCALL_ARG0];  // filename to open
06   ...
07 }
```

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

Details of func - `post_open_hook`

```
01 static void post_open_hook(syscall_ctx_t *ctx) {
02   ...
03   if (unlikely((int)ctx->ret < 0)) {
04     return;
05   }
06   if (strstr(fname, ".so") || strstr(fname, ".so.")) {
07     return;
08   }
09   ...
10 }
```

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`

```
01 static void post_open_hook(syscall_ctx_t *ctx) {
02   ...
03   if (!fd2color[fd]) {
04     color = next_color;
05     fd2color[fd] = color;
06     if (next_color < MAX_COLOR) next_color <<= 1;  // static variable
07   } else {
08     color = fd2color[fd];  // reuse color of a file with the same fd
09   }
10   ...
11 }
```

- "color" can be reused
  - if a file discriptor is closed and then the same file discriptor 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`

```
01 static void post_open_hook(syscall_ctx_t *ctx) {
02   ...
03   if (color2fname[color].empty())
04     color2fname[color] = std::string(fname);
05   else
06     color2fname[color] += " | " + std::string(fname);
07 }
```

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

Details of func - `post_read_hook`

```
01 static void post_read_hook(syscall_ctx_t *ctx) {
02   int fd     =    (int)ctx->arg[SYSCALL_ARG0];
03   void *buf  =  (void*)ctx->arg[SYSCALL_ARG1];
04   size_t len = (size_t)ctx->ret;
05   uint8_t color;
06   ...
07 }
```

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

Details of func - `post_read_hook`

```
01 static void post_read_hook(syscall_ctx_t *ctx) {
02   ...
03   if(unlikely(len <= 0)) {
04     return;
05   }
06
07   fprintf(stderr, "(dta-dataleak) read: %zu bytes from fd %u\n", len, fd);
08   ...
09 }
```

- 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`

```
01 static void post_read_hook(syscall_ctx_t *ctx) {
02   ...
03   color = fd2color[fd];
04   if(color) {
05     tagmap_setn((uintptr_t)buf, len, color);
06   } else {
07     tagmap_clrn((uintptr_t)buf, len);
08   }
09 }
```

- 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`

```
01 static void pre_socketcall_hook(syscall_ctx_t *ctx) {
02   int fd;
03   void *buf;
04   size_t i, len;
05   uint8_t tag;
06   uintptr_t start, end, addr;
07
08   int call             =            (int)ctx->arg[SYSCALL_ARG0];
09   unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];
10
11   switch(call) {
12   case SYS_SEND:
13   case SYS_SENDTO:
14     ...
15     break;
16
17   default:
18     break;
19   }
20 }
```

Details of func - `pre_socketcall_hook`

```
01 static void pre_socketcall_hook(syscall_ctx_t *ctx) {
02   ...
03   int call              =            (int)ctx->arg[SYSCALL_ARG0];
04   unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];
05  ...
06 }
```

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

Details of func - `pre_socketcall_hook`

```
01 static void pre_socketcall_hook(syscall_ctx_t *ctx) {
02   ...
03   switch(call) {
04   case SYS_SEND:
05   case SYS_SENDTO:
06     ...  // omitted
07     break;
08
09   default:
10     break;
11   }
12 }
```

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

Details of func - `pre_socketcall_hook` (omitted part)

```
01 static void pre_socketcall_hook(syscall_ctx_t *ctx) {
02   ...
03     fd  =    (int)args[0];
04     buf =  (void*)args[1];
05     len = (size_t)args[2];
06
07     start = (uintptr_t)buf;
08     end   = (uintptr_t)buf+len;
09     for(addr = start; addr <= end; addr++) {
10       tag = tagmap_getb(addr);
11       if(tag != 0) alert(addr, tag);
12     }
13   ...
14   }
15 }
```

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

Functions

```
01 void alert(uintptr_t addr, uint8_t tag);
02 static void post_open_hook(syscall_ctx_t *ctx);
03 static void post_read_hook(syscall_ctx_t *ctx);
04 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`.

Overview

1. about `libdft`
2. Using DTA to Detect Remote Control-Hijacking
3. Circemeventing 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

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

```
01 int main(int argc, char *argv[]) {
02   size_t i, j, k;
03   FILE *fp[10];
04   char buf1[4096], buf2[4096], *filenames[10];
05   struct sockaddr_storage addr;
06
07   srand(time(NULL));  // set seed
08
09   int sockfd = open_socket("localhost", "9999");  // 1
10
11   socklen_t addrlen = sizeof(addr);
12   // 2
13   recvfrom(sockfd, buf1, sizeof(buf1), 0, (struct sockaddr*)&addr, &addrlen);
14   ...
15 }
```

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

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

```
01 int main(int argc, char *argv[]) {
02   ...
03   size_t fcount = split_filenames(buf1, filenames, 10);  // 1
04   for(i = 0; i < fcount; i++) {
05     fp[i] = fopen(filenames[i], "r");  // 2
06   }
07   i = rand() % fcount;  // 3
08   do { j = rand() % fcount; } while(j == i);
09
10   memset(buf1, '\0', sizeof(buf1));  // initialize buffer
11   memset(buf2, '\0', sizeof(buf2));
12   ...
13 }
```

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`

```
01 int main(int argc, char *argv[]) {
02   ...
03   while(fgets(buf1, sizeof(buf1), fp[i]) && fgets(buf2, sizeof(buf2), fp[j])) {
04
05     for(k = 0; k < sizeof(buf1)-1 && k < sizeof(buf2)-1; k++) {
06       buf1[k] ^= buf2[k];
07     }
08     sendto(sockfd, buf1, strlen(buf1)+1, 0, (struct sockaddr*)&addr, addrlen);
09   }
10
11   return 0;
12 }
```

- Reads each files line by line, concatinating 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/.../dta-execve.cpp .../dta-dataleak.cpp .../date.c .../echo.c
```

1. Runs `dataleak-test-xor` server with `dta-dataleak` as Pin tool,
    - immeediately starting `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

```
01 ...
02 (dta-dataleak) opening /home/binary/.../dta-execve.cpp at fd 5 with color 0x01
03 (dta-dataleak) opening /home/binary/.../dta-dataleak.cpp at fd 6 with color 0x02
04 (dta-dataleak) opening /home/binary/.../date.c at fd 7 with color 0x04
05 (dta-dataleak) opening /home/binary/.../echo.c at fd 8 with color 0x08
06 ...
```

1. Assigns each of files with a taint color.

Test of Data Exfiltration

```
01 ...
02 (dta-dataleak) read: 4096 bytes from fd 6
03 (dta-dataleak) tainting bytes 0x9b775c0 -- 0x9b785c0 with color 0x2
04 (dta-dataleak) read: 155 bytes from fd 8
05 (dta-dataleak) tainting bytes 0x9b785c8 -- 0x9b67663 with color 0x8
06 (dta-dataleak) send: 20 bytes to fd 4
07 ...
```

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

```
01 ...
02 (dta-dataleak) checking taint on bytes 0xffb48f7c -- 0xffb48f90...
03 (dta-dataleak) !!!!!!! ADDRESS 0xffb48f7c IS TANTED (tag=0x0a), ABORTING !!!!!!!
04   tainted by color = 0x02 (/home/binary/code/chapter11/dta-dataleak.cpp)
05   tainted by color = 0x08 (/home/binary/code/chapter11/echo.c)
06 ^C
```

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