Practical Binary Analysis

Chapter 11

山本 航平

11章でやること

- `libdft` の概要
- `libdft` の使い方の例
  - remote control-hijacking attacks を防ぐツール
  - データ漏洩を自動的に検出するツール

11.1 Introducing libdft

`libdft`

- Pin上で構築されたバイト粒度のテイント追跡システム
- DTAライブラリで最も使いやすいものの１つ
- 32-bit x86 のみに対応
- 通常の x86 命令のみに対応
  - MMX や SSE などの命令には非対応
- オープンソース
  - 64-bit への対応, より多くの命令への対応なども可能

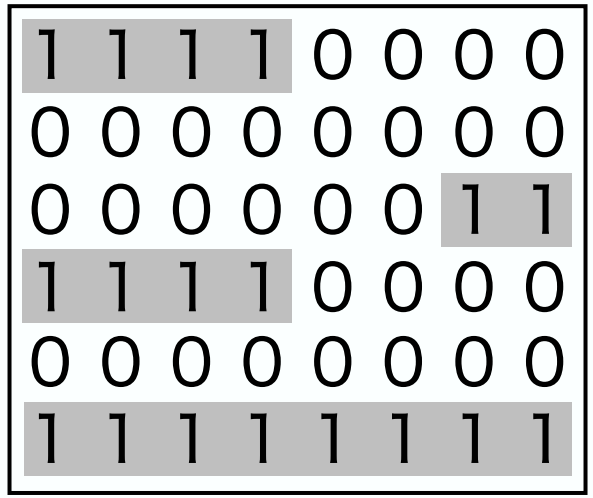
Shadow Memory

- `libdft` には2種類あり、それぞれ異なるシャドウメモリ (tagmap) を使う

① Bitmap

- テイントカラーは1色
- 速い
- メモリのオーバーヘッドが少ない

Bitmap (1 color)



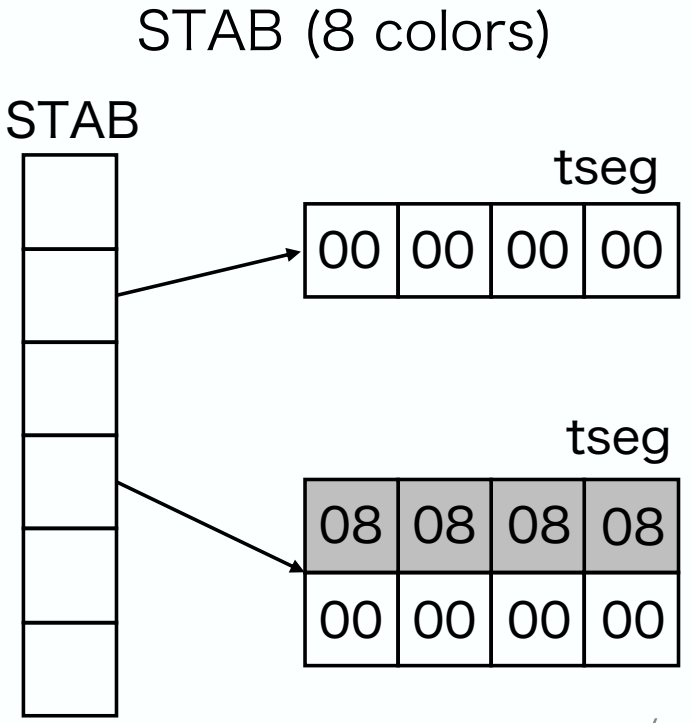
Shadow Memory

② STAB (segment translation table)

・テイントカラーは 8色

・メモリページごとに 1つのエントリ

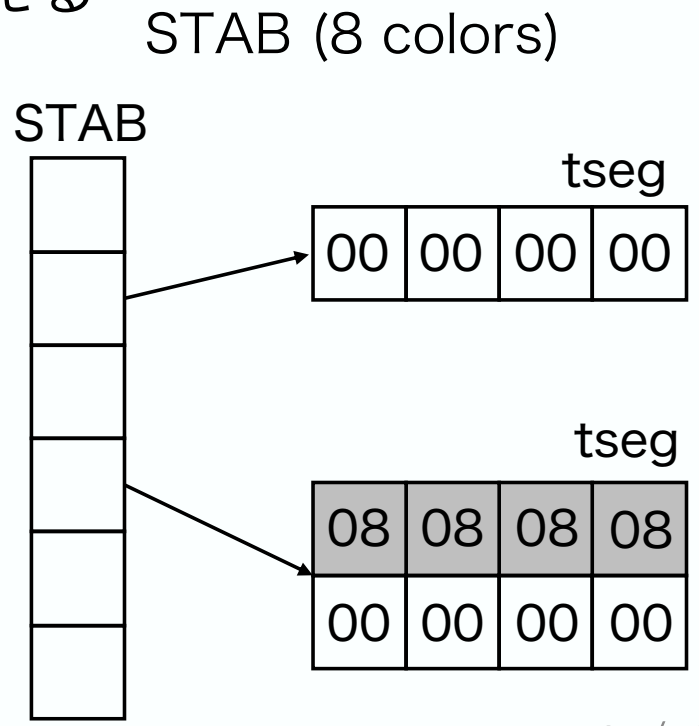
・1つのエントリはシャドウバイトのアドレスを
得るために仮想メモリに足す値(addend)を持つ
ex.) 仮想アドレス : 0x1000
addend : 438
シャドウバイトのアドレス : 0x1438



Shadow Memory

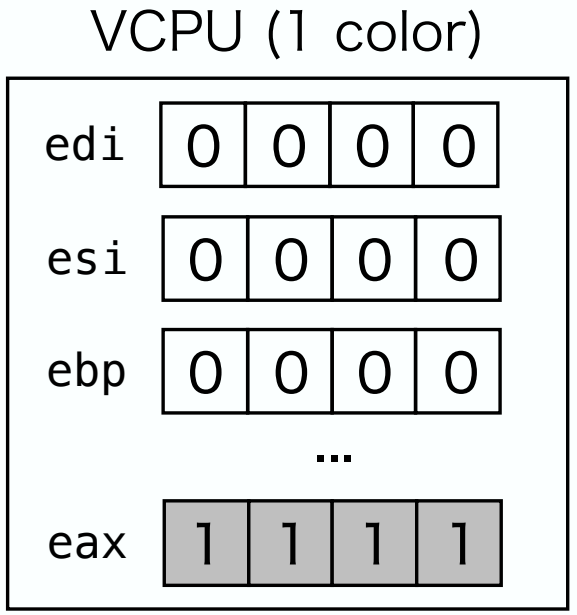
② STAB (segment translation table)

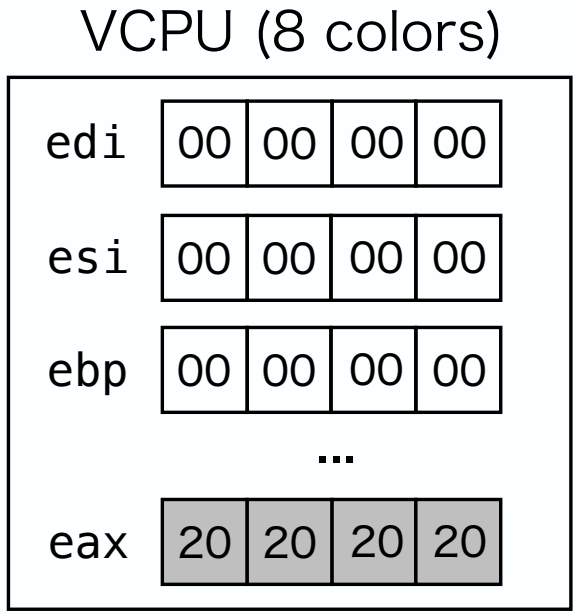
- シャドウメモリは要求された時にページサイズで確保
- ページ内のすべてのアドレスに同じ addend を使える
- 複数のページが隣接した仮想メモリの領域では
  シャドウメモリページも隣接するようにする
- tagmap segment (tseg) :
  隣接したシャドウメモリページのかたまり



Virtual CPU

- CPU レジスタのテイント状態を追跡する
- x86 で利用可能な 32-bit 汎用レジスタ
  (`edi,esi,ebp,esp,ebx,edx,ecx,eax`)
  ごとに4ビットのシャドウメモリ





The libdft API and I/O Interface

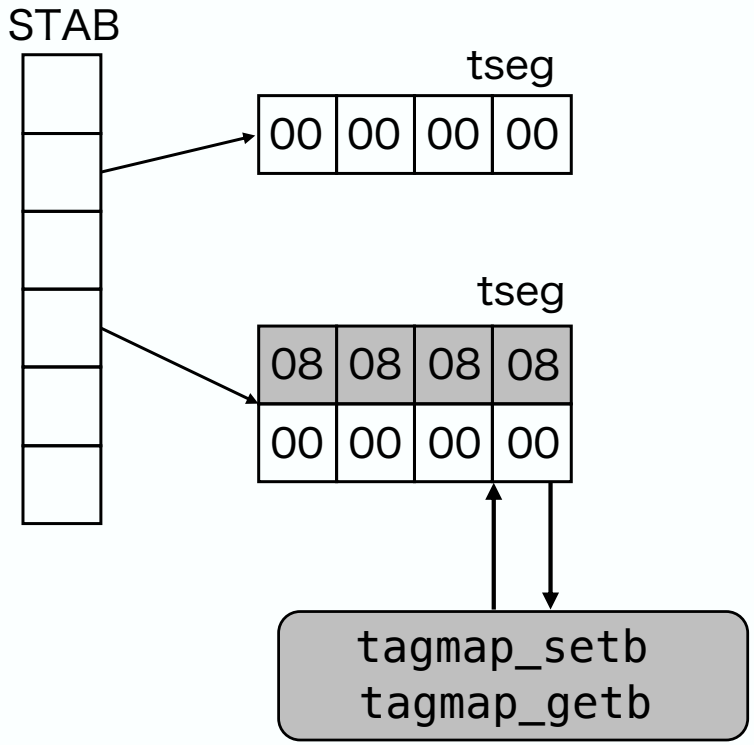
tagmap を操作する API

- `tagmap_setb()`

  メモリバイトをテイントする関数

- `tagmap_getb()`

  メモリバイトのテイント情報を取得する関数



The libdft API and I/O Interface

コールバックと計装コードを追加する API

- `syscall_set_pre(), syscall_set_post()`

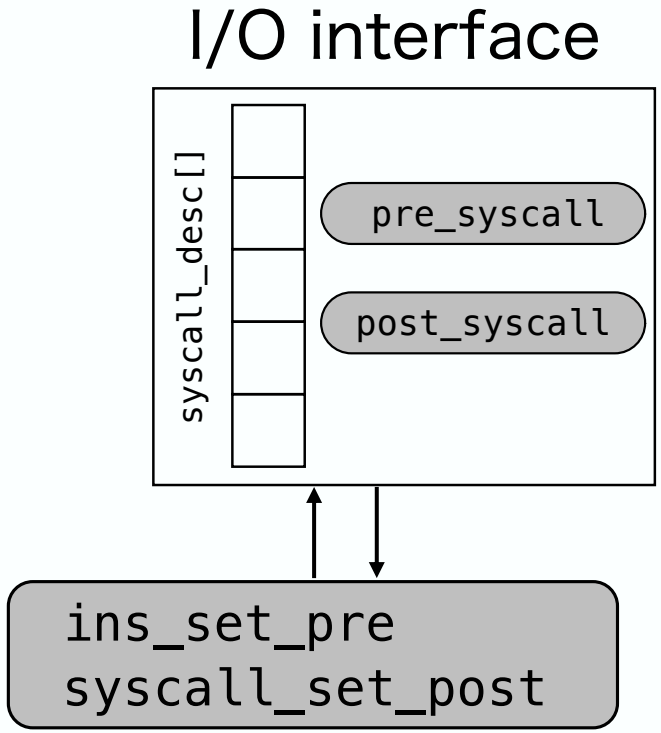
  システムコールイベントのコールバックを登録する関数

- `syscall_desc[]`

  登録したコールバックを保存する配列

- `ins_set_pre(), ins_set_post()`

  命令コールバックを登録する関数



11.1.2 Taint Policy

・ALU

算術演算と論理演算（add, sub, and, xor, div, imul など）

Q. c = a + b → テイント： c = [ ]

・XFER

別のレジスタやメモリに値をコピーするもの（mov など）

Q. c = a → テイント： c = [ ]

・CLR

常に計算結果がテイントされないもの （xor %eax %eax など）

Q. c = a xor a → テイント： c = [ ]

11.1.2 Taint Policy

- SPECIAL

  特別なルールを持つ命令

  `xchg, cmpxchg` : 2つのオペランドのテイントを入れ替える

  `lea` : メモリアドレスからテイントを決定する

  Q. `xchg a b` → テイント： `a` = □, `b` = □

- FPU, MMX, SSE

  `libdft` が対応していない命令

  この命令のテイント追跡はできないため undertainting となる

11.2 Using DTA to Detect Remote Control-Hijacking

dta-execve

- ネットワークで受け取ったデータから execve の引数を操作する攻撃を検出する
- Taint source : ネットワーク受信機能 (recv, recvfrom)
  Taint sink : execve
- 現実的には多くの種類の攻撃を避けるために、他にも taint source や taint sink を定義する必要がある

11.2 Using DTA to Detect Remote Control-Hijacking

```
1 static void post_socketcall_hook (syscall_ctx_t *ctx);
```

- socketcall の後に呼ばれる
  - socketcall : ソケットに関連するイベント ex.) recv, recvfrom
- ネットワークから受け取ったバイトをテイント

```
1 static void pre_execve_hook (syscall_ctx_t *ctx);
```

- execve の前に呼ばれる
- execve の引数のテイントをチェック
- テイントされていればプロセスを終了

11.2 Using DTA to Detect Remote Control-Hijacking

pre_execve_hook() で使う関数

```
1 void check_string_taint(const char *str, const char *source);
```

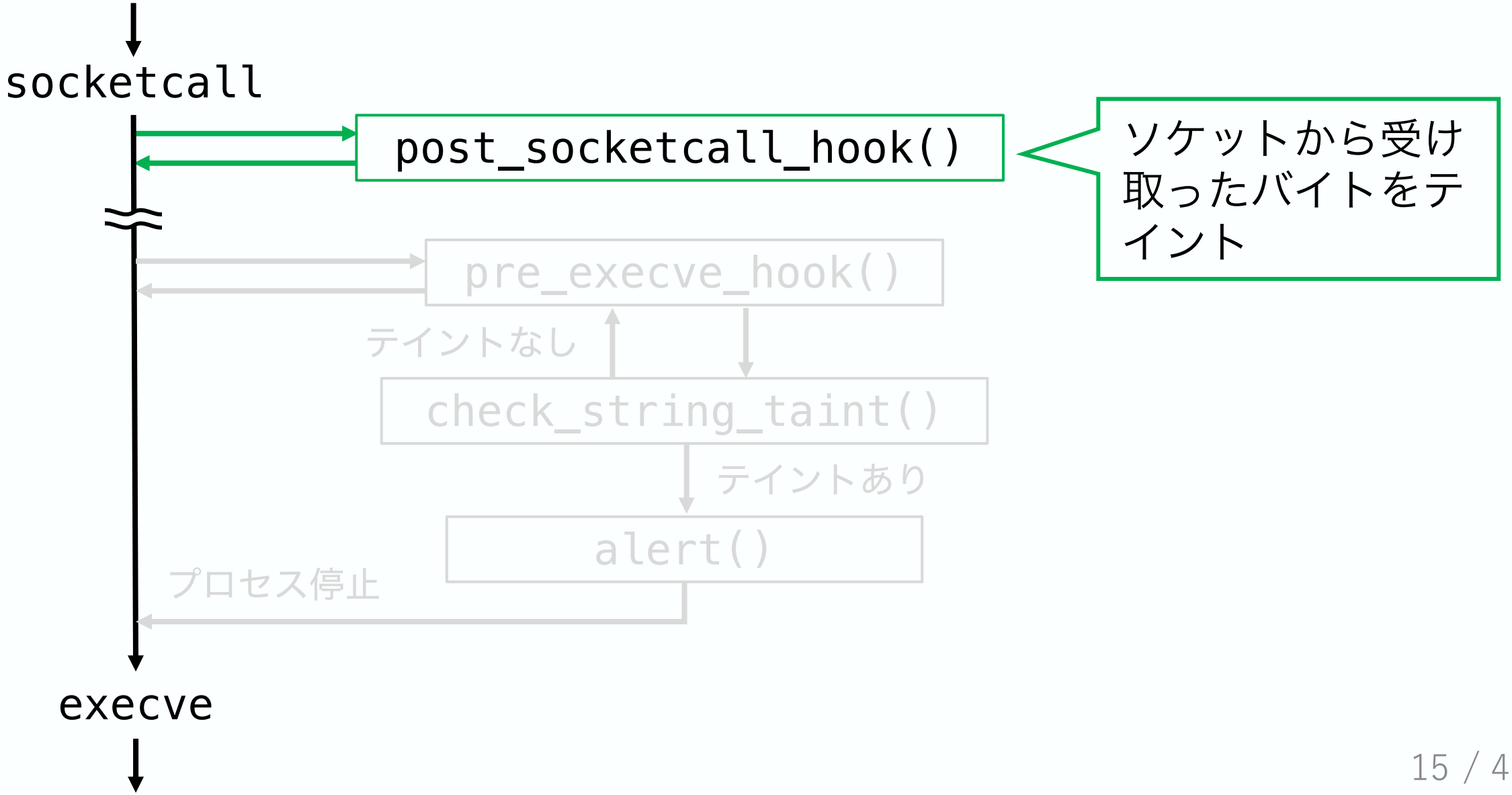
・execve の引数のテイントをチェックする関数

```
1 void  alert(uintptr_t addr, const char *source, uint8_t tag);
```

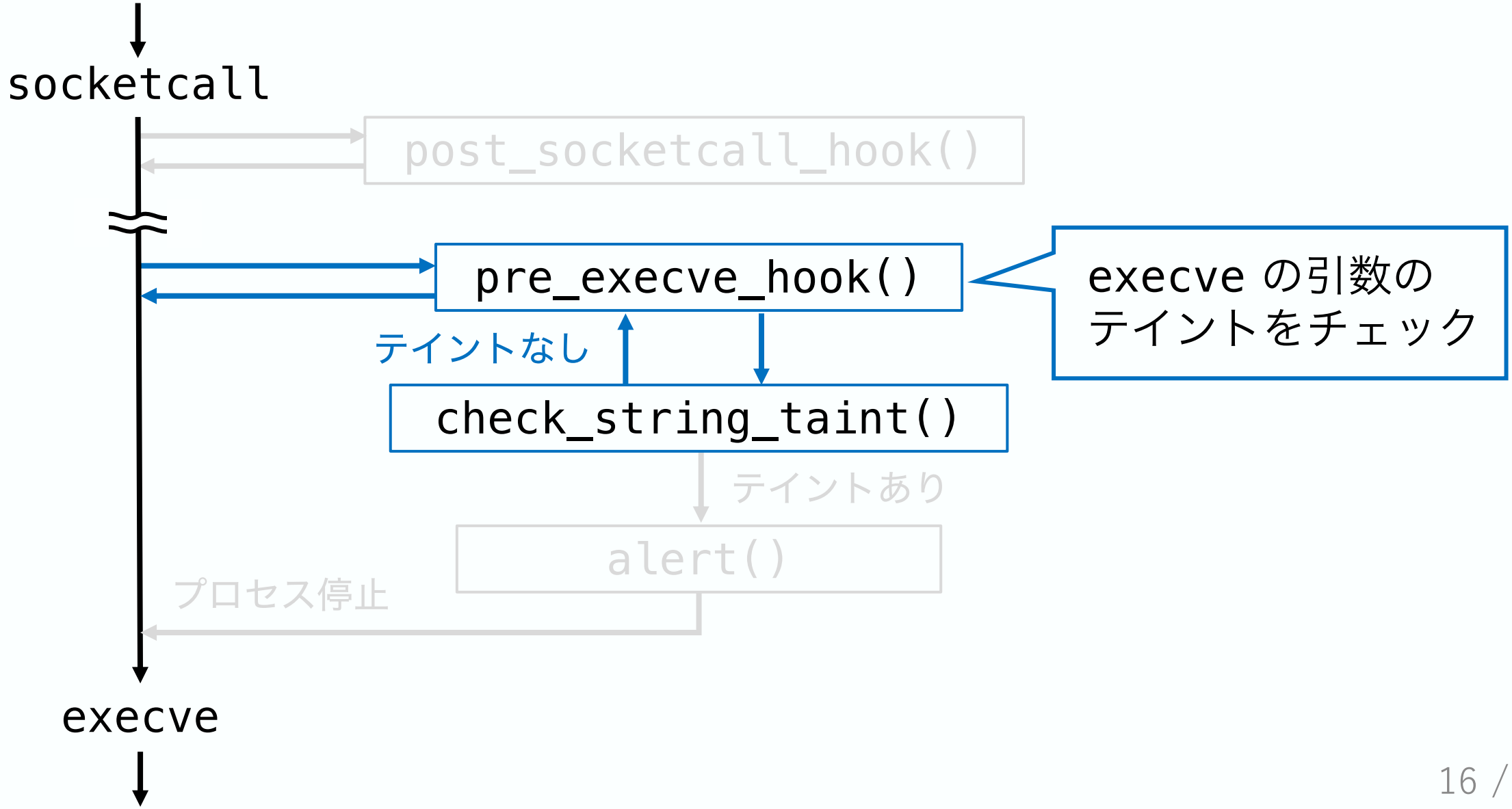
・攻撃を検知したときに呼ばれる関数

・テイントされたアドレスの詳細を出力し、プログラムを終了

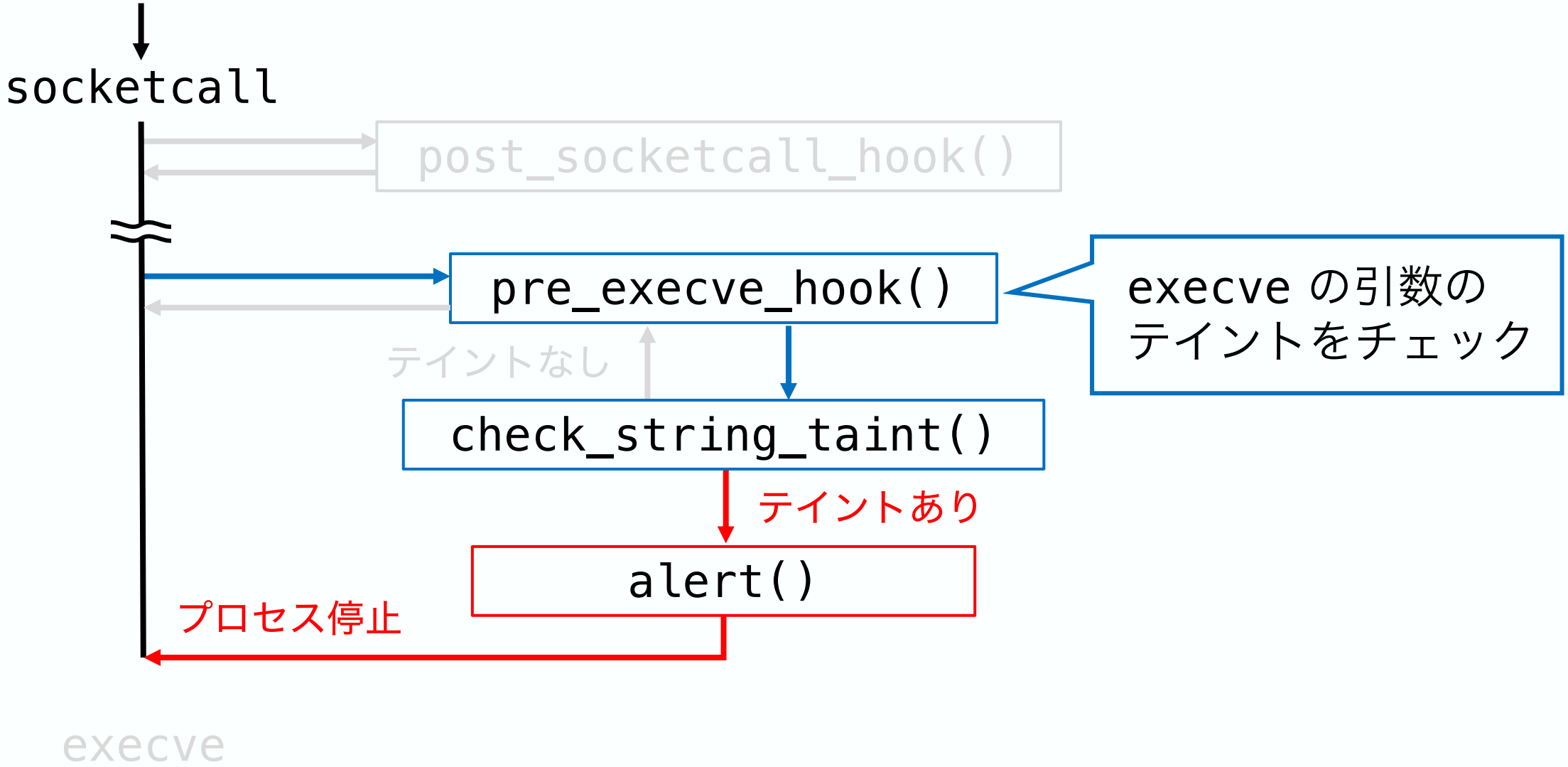
11.2 Using DTA to Detect Remote Control-Hijacking



11.2 Using DTA to Detect Remote Control-Hijacking



11.2 Using DTA to Detect Remote Control-Hijacking



11.2.4 Detecting a Control-Flow Hijacking Attempt

execve-test-overflow

- dta-execve をテストするプログラム
- 想定：

```
foobar: 2022-05-16 16:47:35
```

ソケットから受け取った文字列

date プログラムの実行結果

- 攻撃者のコマンドを実行できる脆弱性を持つ

11.2.4 Detecting a Control-Flow Hijacking Attempt

execve-test-overflow

```
1 static struct __attribute__((packed)) {
2     char prefix[32];
3     char datefmt[32];
4     char cmd[64];
5 } cmd;
```

```
1 for(i = 0; i < strlen(buf); i++) {
2     if(buf[i] == '¥n') {
3         cmd.prefix[i] = '¥0’;
4         break;
5     }
6     cmd.prefix[i] = buf[i];
7 }
```

- ネットワークから受け取った文字列 buf を cmd.prefix にコピー
- 文字数の制限がない
  → 攻撃者が prefix をオーバーフローさせ、datafmt や cmd を上書きすることが可能

A Successful Control Hijack Without DTA

Listing 11-7: Control hijacking in exexve-test-overflow

```
$ ./execve-test-overflow &
[1] 2506
$ nc -u 127.0.0.1 9999
foobar:
(execve-test/child) execv: /home/binary/code/chapter11/date %Y-%m-%d %H:%M:%S
foobar: 2017-12-06 15:25:08
```

- 文字列 "foobar: " をサーバーに送る
- サーバは date コマンドを実行し、"foobar: " と現在日時を出力

A Successful Control Hijack Without DTA

Listing 11-7: Control hijacking in exexve-test-overflow

```
$ ./execve-test-overflow &
[1] 2533
$ nc -u 127.0.0.1 9999
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/code/chapter11/echo
```

- 32文字の A と32文字の B と /home/binary/code/chapter11/echo をサーバに送る

A Successful Control Hijack Without DTA

```
1 static struct __attribute__((packed)) {
2     char prefix[32];
3     char datefmt[32];
4     char cmd[64];
5 } cmd;
```

Q.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/bin
ary/code/chapter11/echo
```

をサーバに送ったとき、

cmd.prefix →

cmd.datefmt →

cmd.cmd →

A Successful Control Hijack Without DTA

Listing 11-7: Control hijacking in exexve-test-overflow

```
1 $ nc -u 127.0.0.1 9999
2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/c
  ode/chapter11/echo
3 (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binar
  y/.../echo
4 AA...AABB...BB/home/binary/code/chapter11/echo BB...BB/home/binary/code/cahpte
  r11/echo
```

- execv は date ではなく echo コマンドを実行
- prefix や datefmt は終端の NULL を含まない
  → prefix, datefmt, cmd の連結文字列として出力

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
1 $ cd /home/binary/libdft/pin-2.13-61206-gcc.4.4.7-linux/
2 $ ./pin.sh -follow_execv -t /home/binary/code/chapter11/dta-execve.so ¥
            -- /home/binary/code/chapter11/execve-test-overflow &
3 [1] 2994
```

- dta-execve を Pin ツールとして Pin を実行
- -follow_execv オプション：
  親プロセスと同様に子プロセスも計装
  execv は子プロセスで呼ばれるため必要

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
$ nc -u 127.0.0.1 9999
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/c
ode/chapter11/echo
(dta-execve) recv: 97 bytes from fd 4
AA...AABB...BB/home/binary/code/chapter11/echo ¥x0a
(dta-execve) tainting bytes 0xffa231ec -- 0xffa2324d with tag 0x1
```

・先程と同様に prefix がオーバーフローし cmd を書き換える文字列をサーバに送る

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
$ nc -u 127.0.0.1 9999
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/c
ode/chapter11/echo
(dta-execve) recv: 97 bytes from fd 4
AA...AABB...BB/home/binary/code/chapter11/echo ¥x0a
(dta-execve) tainting bytes 0xffa231ec -- 0xffa2324d with tag 0x1
```

・socketcall ハンドラが受け取ったすべてのバイトをテイント (post_socketcall_hook)

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
1 (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binar
  y/.../echo
2 (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
3 (dta-execve) checking taint on bytes 0x804b100 0x804b120 (execve command)...
4 (dta-execve) !!!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01),
  ABORTING !!!!!!!
5 AA...AABB...BB/home/binary/code/chapter11/echo
6 [1]+ Done ./pin.sh -follow_execv ...
```

- 攻撃者が仕組んだ echo コマンドが実行されようとしている (exec_cmd)
- dta-execve が execve の実行を妨害 (pre_execve_hook)

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
1 (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binar
  y/.../echo
2 (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
3 (dta-execve) checking taint on bytes 0x804b100 0x804b120 (execve command)...
4 (dta-execve) !!!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01),
  ABORTING !!!!!!!
5 AA...AABB...BB/home/binary/code/chapter11/echo
6 [1]+ Done ./pin.sh -follow_execv ...
```

・execve のすべての引数のテイントをチェック（check_string_taint）

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
1 (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binar
  y/.../echo
2 (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
3 (dta-execve) checking taint on bytes 0x804b100 0x804b120 (execve command)...
4 (dta-execve) !!!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01),
  ABORTING !!!!!!!
5 AA...AABB...BB/home/binary/code/chapter11/echo
6 [1]+ Done ./pin.sh -follow_execv ...
```

・コマンドはテイントされているため、dta-execve はアラートを出し攻撃者のコマンドを実行しようとした子プロセスを停止 (alert)

・これによって攻撃者のコマンドの実行を防ぐことができる

Using DTA to Detect the Hijacking Attempt

Listing 11-8: Detecting an attempted control hijack with dta-execve

```
1 (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binar
  y/.../echo
2 (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
3 (dta-execve) checking taint on bytes 0x804b100 0x804b120 (execve command)...
4 (dta-execve) !!!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01),
  ABORTING !!!!!!!
5 AA...AABB...BB/home/binary/code/chapter11/echo
6 [1]+ Done ./pin.sh -follow_execv ...
```

・execve が実行される前の出力のみが表示

11.3 Circumventing DTA with Implicit Flows

- `libdft` などの DTA システムでは implicit flow を通して伝播されるデータを追跡できない
- 先程のテストコードに implicit flow を含めると攻撃を防ぐことができない
- テイント解析を混乱させるためマルウェアに implicit flow が含まれることがある

11.3 Circumventing DTA with Implicit Flows

Listing 11-9: execve-test-overflow-implicit.c (exec_cmd)

```
1  for(i = 0; i < strlen(buf); i++) {
2      if(buf[i] == '¥n') {
3          cmd.prefix[i] = '¥0';
4          break;
5      }
6      /* cmd.prefix[i] = buf[i] */
7      char c = 0;
8      while(c < buf[i]) c++;
9      cmd.prefix[i] = c;
10 }
```

・buf[i] は implicit な方法 (while ループ) で c にコピーされている

→ buf と prefix は implicit なデータフローしかない

11.4 A DTA-Based Data Exfiltration Detector

`dta-dataleak`

- ファイルの情報漏洩を検知するツール
- 複数のテイントカラーを使って「どの」ファイルから漏洩したかを調べる
- Taint source : `open`, `read`
  Taint sink : ネットワーク送信機能（`send`, `sendto`）

11.4 A DTA-Based Data Exfiltration Detector

```
1 static void post_open_hook (syscall_ctx_t *ctx);
```

・open の後に呼ばれる

・ファイルディスクリプタとテイントカラーを対応づけ

・追跡不要なファイルは無視 (共有ライブラリなど)

```
1 static void post_read_hook (syscall_ctx_t *ctx);
```

・read の後に呼ばれる

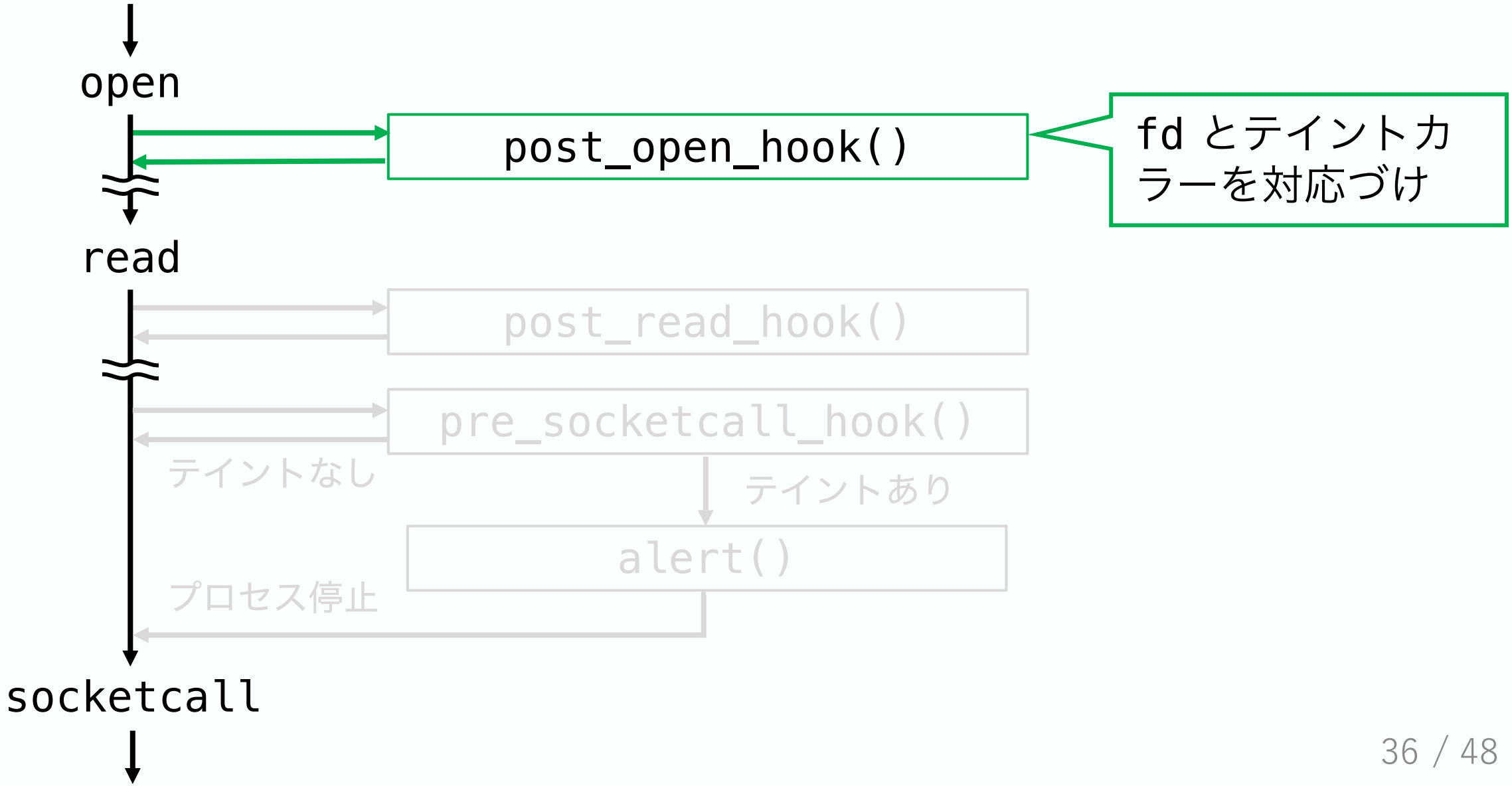
・ファイルから読み込んだバイトをテイント

11.4 A DTA-Based Data Exfiltration Detector

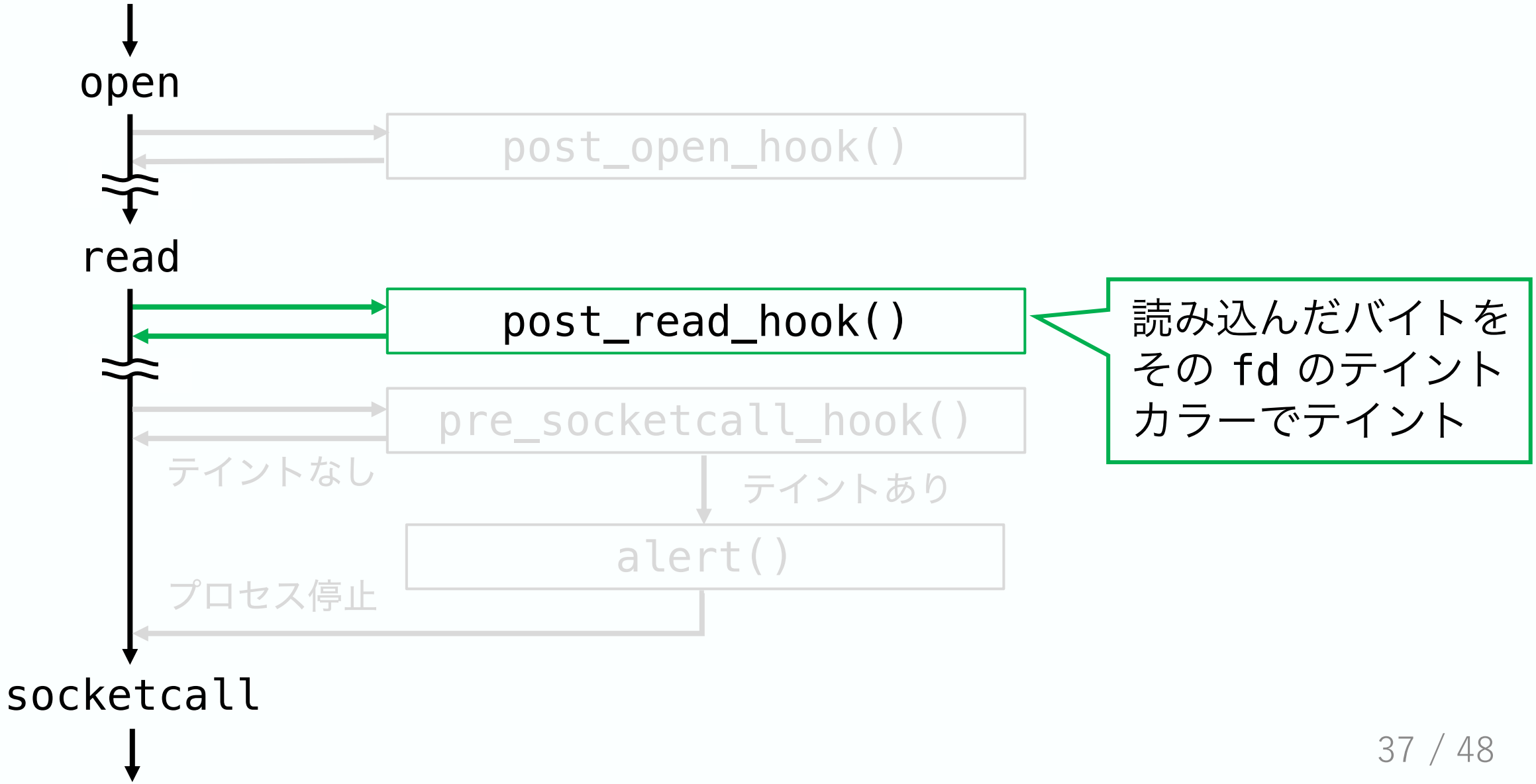
```
1 static void pre_socketcall_hook (syscall_ctx_t *ctx);
```

- `socketcall` の前に呼ばれる
- 送信するバイトのテイントをチェック

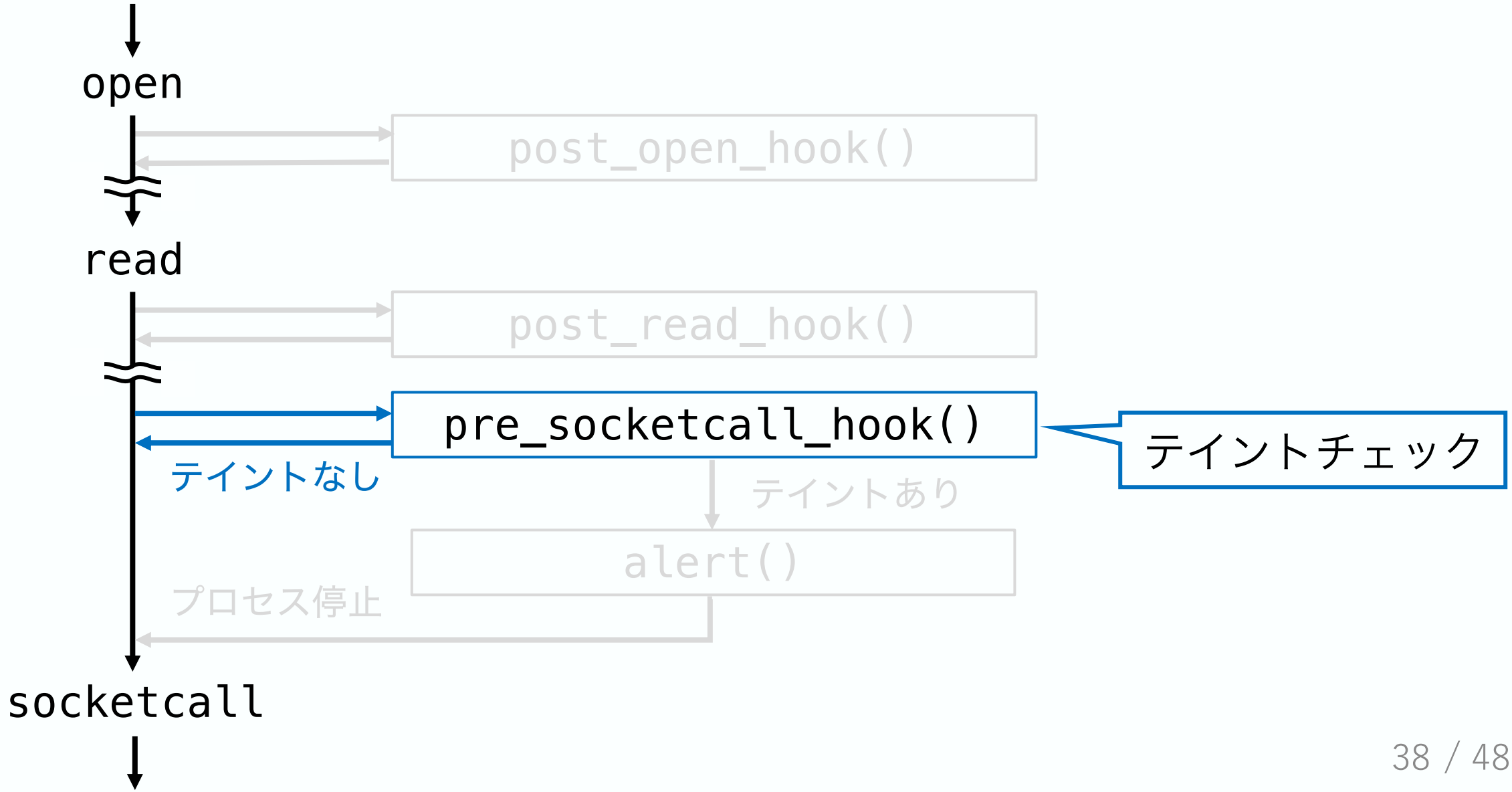
11.4 A DTA-Based Data Exfiltration Detector



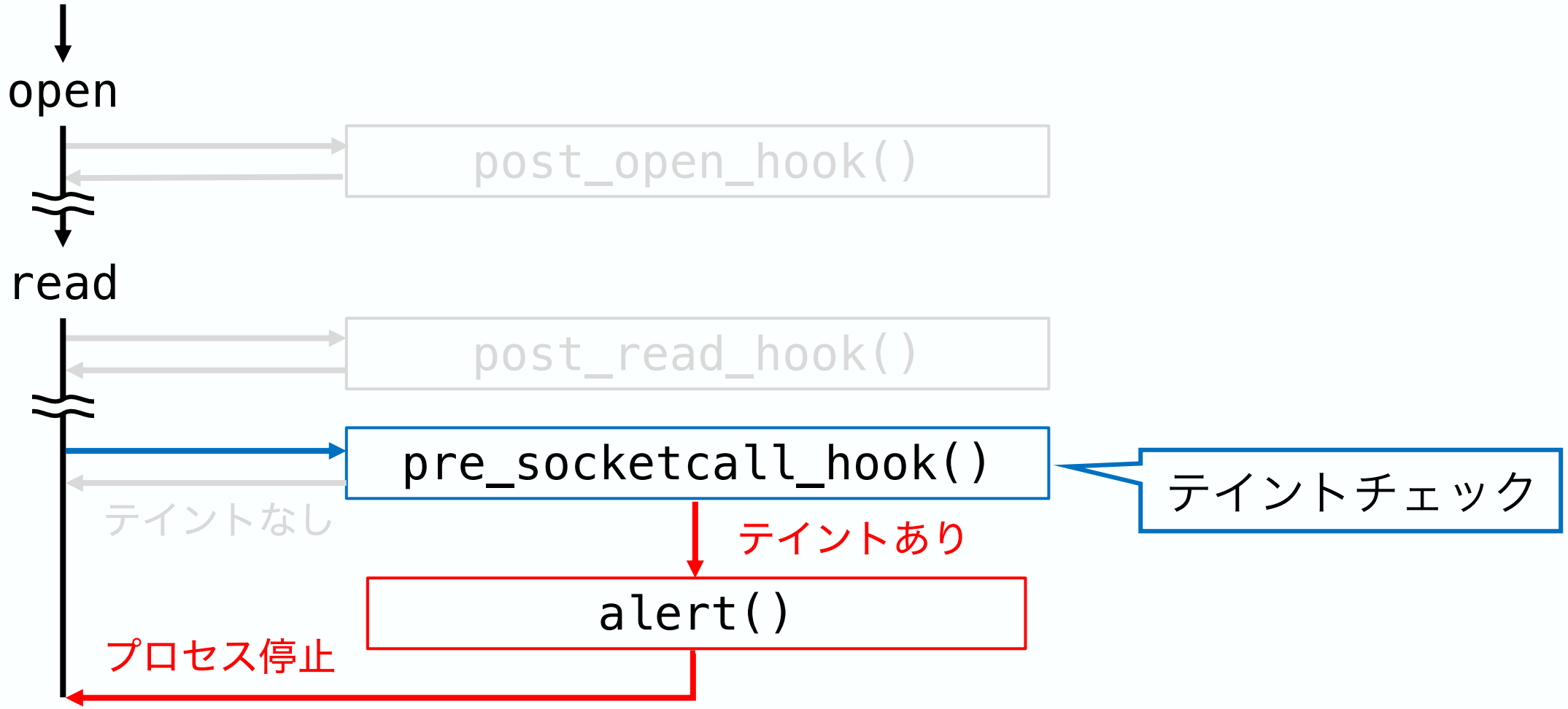
11.4 A DTA-Based Data Exfiltration Detector



11.4 A DTA-Based Data Exfiltration Detector



11.4 A DTA-Based Data Exfiltration Detector



11.4.3 Detecting a Data Exfiltration Attempt

`dataleak-test-xor`

- `dta-dataleak` をテストするためのサーバプログラム

- サーバが自発的にファイルを漏洩させる

  → 現実的にはエクスプロイト(脆弱性を悪用した攻撃)を通して
  ファイルにアクセスされる

- XOR で 2つのファイル内容を結合させてネットワークに送信

  → テイントカラーの結合もテスト可能

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

```
$ cd ~/libdft/pin-2.13-61206-gcc.4.4.7-linux/
$ ./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 oxff8b34d0 Oxff8b36d0
[1] 22713
```

・Pin で dataleak-test-xor サーバを実行し、dta-dataleak をデータ漏洩を防ぐ Pin ツールとして使う

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

```
$ cd ~/libdft/pin-2.13-61206-gcc.4.4.7-linux/
$ ./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 oxff8b34d0 Oxff8b36d0
[1] 22713
```

・すぐに dataleak-test-xor のプロセスを読み込むための read システムコールが発生する

・これらのバイトは共有ライブラリから読み込まれたもの

→ dta-dataleak はこれを無視する

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

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

・netcat セッションを開始しサーバに接続する

・ファイル名のリストをサーバに送信する

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

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

・open イベントの後、ファイルにテイントカラーを割り当てる（post_open_hook）

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

```
(dta-dataleak) read: 155 bytes from fd 8
(dta-dataleak) tainting bytes 0x872a5c0 -- 0x872a65b with color 0x8
(dta-dataleak) read: 3923 bytes from fd 5
(dta-dataleak) tainting bytes 0x872b5c8 -- 0x872c51b with color 0x1
```

・サーバはランダムに 2つのファイルを選ぶ (ここでは８と５)

・read イベントが発生すると読み取ったバイトをそれぞれのテイントカラー(0x08, 0x01)でテイントする (post_read_hook)

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

```
(dta-dataleak) send: 20 bytes to fd 4
¥x0cCdclude <stdio.h>¥x0a¥x00
(dta-dataleak) checking taint on bytes 0xff8b19cc -- 0xff8b19e0...
(dta-dataleak) !!!!!!! ADDRESS 0xff8b19cc IS TAINTED (tag=      ), ABORTING !!!!!!!
tainted by color = 0x01 (/home/binary/code/chapter11/dta-execve.cpp)
tainted by color = 0x08 (/home/binary/code/chapter11/echo.c)
[1]+ Exit 1 ./pin.sh -follow_execv -t ~/code/chapter11/dta-dataleak.so ...
```

・サーバが送信しようとしているバイトのテイントをチェックする (pre_socketcall_hook)

11.4.3 Detecting a Data Exfiltration Attempt

Listing 11-16: Detecting a data exfiltration attempt with dta-dataleak

```
(dta-dataleak) send: 20 bytes to fd 4
¥x0cCdclude <stdio.h>¥x0a¥x00
(dta-dataleak) checking taint on bytes 0xff8b19cc -- 0xff8b19e0...
(dta-dataleak) !!!!!!! ADDRESS 0xff8b19cc IS TAINTED (tag=    ), ABORTING !!!!!!!
tainted by color = 0x01 (/home/binary/code/chapter11/dta-execve.cpp)
tainted by color = 0x08 (/home/binary/code/chapter11/echo.c)
[1]+ Exit 1 ./pin.sh -follow_execv -t ~/code/chapter11/dta-dataleak.so ...
```

・バイトは tag = [redacted] でテイントされている
→ アラートを出しプログラムを終了する

・アラートから 2つのテイントカラーに対応するファイルが
dta-execve.cpp, echo.c だと分かる

11.5 Summary

・DTAライブラリである `libdft` の内部について学んだ

・`libdft` を使って control hijacking と データ漏洩 という２種類の一般的な攻撃を検知する例を見た

実装の詳細

Contents – 11.2

Contents – 11.2

dta-execve – main

Listing 11-1: dta-execve.cpp

```
1 #include "pin.H"
2
3 #include "branch_pred.h"
4 #include "libdft_api.h"
5 #include "syscall_desc.h"
6 #include "tagmap.h”
```

- すべての `libdft` ツールは `libdft` ライブラリに関連づけられた Pin ツール
- `libdft` API にアクセスするために必要

dta-execve – main

Listing 11-1: dta-execve.cpp

```
1 extern syscall_desc_t syscall_desc[SYSCALL_MAX];
```

・システムコールフックの追跡に使う配列

dta-execve – main

Listing 11-1: dta-execve.cpp (main)

```
1  PIN_InitSymbols();
2  if(unlikely(PIN_Init(argc, argv))) {
3      return 1;
4  }
```

- Pin の初期化
- `unlikely()`:
  コンパイラに引数が `true` になる可能性が低いことを伝える
  → コンパイラの分岐予測を助け、高速なコードを生成するかも

dta-execve – main

Listing 11-1: dta-execve.cpp (main)

```
1 if(unlikely(libdft_init() != 0)) {
2     libdft_die();
3     return 1;
4 }
```

- libdft を初期化
- tagmap などのデータ構造を作成する
- 初期化に失敗したときは libdft_die()で割り当てたリソースを解放

dta-execve – main

Listing 11-1: dta-execve.cpp (main)

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

・システムコールが実行されるときに呼ばれるハンドラを設定

・socketcall の後に実行される post_socketcall_hook を設定

- socketcall : ソケットに関連するイベント ex.) recv, recvfrom

・execve の前に実行される pre_execve_hook を設定

dta-execve – main

Listing 11-1: dta-execve.cpp

```
1 extern syscall_desc_t syscall_desc[SYSCALL_MAX];
  ...
2 syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
3 syscall_set_pre (&syscall_desc[__NR_execve], pre_execve_hook);
```

- `syscall_set_post()`, `syscall_set_pre()` の引数
  - ① `syscall_desc` 配列のエントリを指すポインタ
  - ② ハンドラの関数ポインタ

- `syscall_desc` の index はシステムコール番号

dta-execve – main

Listing 11-1: dta-execve.cpp (main)

```
1 PIN_StartProgram();
```

・計装プログラムを実行

・ここから制御が戻ってくることはない

dta-execve – main

```
1  extern ins_desc_t ins_desc[XED_ICLASS_LAST];
   ...
2  ins_set_post(&ins_desc[XED_ICLASS_RET_NEAR], dta_instrument_ret);
```

- 命令もシステムコール同様にフックできる
- `ins_desc` の index は Intel's x86 encoder/decoder library (XED) が提供するシンボル名を使う

Contents – 11.2

dta-execve – alert

Listing 11-2: dta-execve.cpp

```
1 void  alert(uintptr_t addr, const char *source, uint8_t tag);
```

・攻撃を検知したときに呼ばれる関数

・テイントされたアドレスの詳細を出力し、プログラムを終了

dta-execve – check_string_taint

Listing 11-2: dta-execve.cpp

```
1 void check_string_taint(const char *str, const char *source);
```

- execve の引数のテイントをチェックする関数
- str: テイントをチェックする文字列

  source: str がどこから来たものかを判別する文字列
  (execve のパス or execve のパラメータ or
  環境変数のパラメータ)

dta-execve — check_string_taint

Listing 11-2: dta-execve.cpp (check_string_taint)

```
1 uint8_t tag;
2 uintptr_t start = (uintptr_t)str;
3 uintptr_t end = (uintptr_t)str+strlen(str);
4
5 for(uintptr_t addr = start; addr <= end; addr++) {
6     tag = tagmap_getb(addr);
7     if(tag != 0) alert(addr, source, tag);
8 }
```

- `str` のバイトごとにテイントを調べる
- `tagmap_getb(addr)`: `addr` のテイントカラーを含んだシャドウバイトを返す
- `tag != 0` → そのバイトはテイントされている → 攻撃検知

Contents – 11.2

dta-execve – post_socketcall_hook

Listing 11-1: dta-execve.cpp (main)

```
1 syscall_set_post (&syscall_desc[__NR_socketcall],
                                  post_socketcall_hook);
```

Listing 11-3: dta-execve.cpp

```
1 static void post_socketcall_hook (syscall_ctx_t *ctx);
```

- post_socketcall_hook():
  socketcall システムコールの後に呼ばれ、ネットワークから受け取ったバイトをテイントする

- 引数 ctx: システムコールの引数や返値の情報を含む

dta-execve – post_socketcall_hook

Listing 11-3: dta-execve.cpp (post_socketcall_hook)

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

- socketcall は 2つの引数を持つ
  - call: recv や recvfrom など、どのソケット関数かを表す
  - arg: 実際のシステムコールに渡される引数のブロック

dta-execve – post_socketcall_hook

Listing 11-3: dta-execve.cpp (post_socketcall_hook)

```
1 switch(call) {
2 case SYS_RECV:
3 case SYS_RECVFROM:
4     ...
5     break;
6 default:
7     break;
```

- socketcall の種類が recv か recvfrom のとき受け取ったバイトをテイントする必要がある
- それ以外の場合は何もしない

dta-execve – post_socketcall_hook

Listing 11-3: dta-execve.cpp (post_socketcall_hook)

```
1 if(unlikely(ctx->ret <= 0)) {
2     return;
3 }
```

- ソケットの返値が 0 以下のとき、何のバイトも受け取っていない
  → 何もする必要がない

- この返値はシステムコール実行の後でしか取得できない

dta-execve – post_socketcall_hook

Listing 11-3: dta-execve.cpp (post_socketcall_hook)

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

・args: ソケット関数の引数が同じ順番で入る

recv, recvfrom:

arg[0]: ソケットファイルディスクリプタ

arg[1]: 受け取ったバッファアドレス

・ctx->ret: 受け取ったバイトの長さ

dta-execve – post_socketcall_hook

Listing 11-3: dta-execve.cpp (post_socketcall_hook)

```
1 tagmap_setn((uintptr_t)buf, len, 0x01);
```

・受け取ったバイトを複数同時にテイントする

buf: テイントするバイトの先頭アドレス

len: テイントするバイト数

0x01: テイントカラー

Contents – 11.2

dta-execve – pre_execve_hook

Listing 11-1: dta-execve.cpp (main)

```
1 syscall_set_pre (&syscall_desc[__NR_execve],
                                    pre_execve_hook);
```

Listing 11-4: dta-execve.cpp

```
1 static void pre_execve_hook (syscall_ctx_t *ctx);
```

- pre_execve_hook():
  execve システムコールの前に呼ばれ、引数がテイントされていないことを確認する

- 引数 ctx: システムコールの引数や返値の情報を含む

dta-execve – pre_execve_hook

Listing 11-4: dta-execve.cpp (pre_execve_hook)

```
1 const char *filename = (const char*)ctx->arg[SYSCALL_ARG0];
2 char * const *args = (char* const*)ctx->arg[SYSCALL_ARG1];
3 char * const *envp = (char* const*)ctx->arg[SYSCALL_ARG2];
```

- execve は 3つの引数を持つ
  - filename: execve が実行するプログラムのファイル名
  - args: 新しいプログラムの引数の配列
  - envp: 新しいプログラムの環境変数の配列
- この 3つの引数どれでもテイントされていれば、アラートを出すようにする

dta-execve – pre_execve_hook

Listing 11-4: dta-execve.cpp (pre_execve_hook)

```
1 check_string_taint(filename, "execve command");
2 while(args && *args) {
3     check_string_taint(*args, "execve argument");
4     args++;
5 }
6 while(envp && *envp) {
7     check_string_taint(*envp, "execve environment parameter");
8     envp++;
9 }
```

- execve のすべての引数について check_string_taint() で検査
  → テイントされている引数があれば、アラートを出しプログラム終了

Contents – 11.2

execve-test-overflow – main

Listing 11-5: execve-test-overflow.c (main)

```
1 char buf[4096];
2 struct sockaddr_storage addr;
3
4 int sockfd = open_socket("localhost", "9999");
5
6 addrlen = sizeof(addr);
7 recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr*)&addr, &addrlen);
8
9 int child_fd = exec_cmd(buf);
```

- ソケットを開く
- recvfrom システムコールでソケットからのメッセージを buf に入れる
- exec_cmd(buf) を実行

execve-test-overflow - main

Listing 11-5: execve-test-overflow.c (main)

```
1 int child_fd = exec_cmd(buf);
2 FILE *fp = fdopen(child_fd, "r");
3
4 while(fgets(buf, sizeof(buf), fp)) {
5     sendto(sockfd, buf, strlen(buf)+1, 0, (struct sockaddr*)&addr, addrlen);
6 }
```

・実行されたコマンドの出力をネットワークソケットに書き出す

execve-test-overflow

exec_cmd(char *buf)

- 現在日時を取得する date プログラムを実行し、その後サーバーは結果をネットワークに出力する

- 想定：

```
foobar: 2022-05-16 16:47:35
```

ソケットから受け取った文字列（buf）

date プログラムの実行結果

execve-test-overflow

Listing 11-6: execve-test-overflow.c

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

・コマンドとパラメータを保存する構造体

`prefix:` コマンド出力 (ソケットから受け取った文字列)

`datefmt:` 出力するデータフォーマット

`cmd:` 実行するプログラム

Linux デフォルトの `date` プログラムは 64-bit なので `libdft` が対応していない

execve-test-overflow – exex_cmd

Listing 11-6: execve-test-overflow.c (exec_cmd)

```
1 for(i = 0; i < strlen(buf); i++) {
2     if(buf[i] == '¥n') {
3         cmd.prefix[i] = '¥0’;
4         break;
5     }
6     cmd.prefix[i] = buf[i];
7 }
```

・ネットワークから受け取った文字列 buf を cmd.prefix にコピー

・文字数の制限がない

→ 攻撃者が prefix をオーバーフローさせ、datafmt や cmd を上書きすることが可能

execve-test-overflow – exex_cmd

Listing 11-6: execve-test-overflow.c (exec_cmd)

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

・cmd と datafmt を execv で使う argv 配列にコピー

execve-test-overflow – exex_cmd

Listing 11-6: execve-test-overflow.c (exec_cmd)

```
 1 pipe(p);
 2 switch(pid = fork()) {
 3 case 0: /* Child */
 4     close(1);
 5     dup(p[1]);
 6     close(p[0]);
 7
 8     printf("%s", cmd.prefix);
 9     fflush(stdout);
10     execv(argv[0], argv);
11     perror("(execve-test/child) execve failed");
12     kill(getppid(), SIGINT);
13     exit(1);
```

```
1 default: /* parent */
2     close(p[1]);
3     return p[0];
4 }
```

- 子プロセスで execv を実行
- パイプを通して親プロセスが子プロセスの出力を読み取る

Contents – 11.4

Contents – 11.4

dta-dataleak

Listing 11-10: dta-dataleak.c

```
1 static std::map<int, uint8_t> fd2color;
2 static std::map<uint8_t, std::string> color2fname;
3
4 #define MAX_COLOR 0x80
```

- fd2color:
  ファイルディスクリプタからテイントカラーへのmap
- color2fname:
  テイントカラーからファイル名へのmap
- MAX_COLOR:
  テイントカラーの最大値

dta-dataleak – main

Listing 11-10: dta-dataleak.c (main)

```
1 syscall_set_post(&syscall_desc[__NR_open], post_open_hook);
2 syscall_set_post(&syscall_desc[__NR_read], post_read_hook);
3 syscall_set_pre (&syscall_desc[__NR_socketcall], pre_socketcall_hook);
```

- open, read の後に実行される post_open_hook, post_read_hook を設定
- socketcall の前に実行される post_socketcall_hook を設定

dta-dataleak – alert

Listing 11-11: dta-dataleak.c (alert)

```
1  fprintf(stderr, "¥n(dta-dataleak) !!!!!!! ADDRESS 0x%x IS TAINTED
   (tag=0x%02x), ABORTING !!!!!!!¥n", addr, tag);
2
3  for(unsigned c = 0x01; c <= MAX_COLOR; c <<= 1) {
4      if(tag & c) {
5          fprintf(stderr, " tainted by color = 0x%02x (%s)¥n",
6                                              c, color2fname[c].c_str());
7      }
8  }
9  exit(1);
```

- どのアドレスがどの色でテイントされているかを表示
- テイントカラーに対応するファイル名を color2fname から読み取る
- プログラムを終了しデータの漏洩を防ぐ

Contents – 11.4

dta-dataleak – post_open_hook

Listing 11-12: dta-dataleak.cpp

```
1 static void post_open_hook (syscall_ctx_t *ctx);
```

- open システムコールの後に呼ばれる
- ファイルが開かれたときにそのファイルディスクリプタにテイントカラーを割り当てる
- 共有ライブラリなどテイントする必要のないファイルは除外する

dta-dataleak — post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 static uint8_t next_color = 0x01;
2 uint8_t color;
3 int fd             =          (int)ctx->ret;
4 const char *fname = (const char*)ctx->arg[SYSCALL_ARG0];
5
6 if(unlikely((int)ctx->ret < 0)) {
7     return;
8 }
```

- next_color: 次に利用可能なテイントカラー
- fd: 開いたファイルのファイルディスクリプタ
- fname: 開いたファイルのファイル名

dta-dataleak – post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 if(strstr(fname, ".so") || strstr(fname, ".so.")) {
2     return;
3 }
```

・開いたファイルが追跡する必要のない共有ファイルのとき、そのファイルにテイントカラーを割り当てない

・ここではファイル拡張子から共有ライブラリかどうかを判断する

dta-dataleak – post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 if(!fd2color[fd]) {
2     color = next_color;
3     fd2color[fd] = color;
4     if(next_color < MAX_COLOR) next_color <<= 1;
5 }
```

・開いたファイルのディスクリプタにテイントカラーが割り当てられていないとき

- そのファイルディスクリプタに new_color を割り当てる
- new_color のビットを左に1ずらす

ex.) 0x01 → 0x02

dta-dataleak — post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 if(!fd2color[fd]) {
2     color = next_color;
3     fd2color[fd] = color;
4     if(next_color < MAX_COLOR) next_color <<= 1;
5 }
```

- libdft は 8色までしか対応していない (MAX_COLOR = 0x80)
- next_color が 0x80 に到達した後に開かれたファイルのテイントカラーはすべて 0x80 になる
- カラー 0x80 に一致するファイルは一つとは限らない

dta-dataleak – post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 if(!fd2color[fd]) {
2     ...
3 } else {
4     color = fd2color[fd];
5 }
```

・以前開かれたファイルが閉じられ、別のファイルを開いたときに同じファイルディスクリプタが割り当てられたとき

- 既に割り当てられたテイントカラーを使用する
- 簡単のためこのカラーに一致するファイルもリストにする

dta-dataleak – post_open_hook

Listing 11-12: dta-dataleak.cpp (post_open_hook)

```
1 if(color2fname[color].empty()) color2fname[color] = std::string(fname);
2 else color2fname[color] += " | " + std::string(fname);
```

・各テイントカラーに一致するファイル名を保存

・テイントカラーが再利用された場合は " | " で繋げたリストにする

Contents – 11.4

dta-dataleak – post_read_hook

Listing 11-13: dta-dataleak.cpp

```
1 static void post_read_hook (syscall_ctx_t *ctx);
```

- read システムコールの後に呼ばれる
- ファイルから読まれたバイトをそのファイルのテイントカラーでテイントする

dta-dataleak – post_read_hook

Listing 11-13: dta-dataleak.cpp (post_read_hook)

```
1 int fd = (int)ctx->arg[SYSCALL_ARG0];
2 void *buf = (void*)ctx->arg[SYSCALL_ARG1];
3 size_t len = (size_t)ctx->ret;
4
5 if(unlikely(len <= 0)) {
6     return;
7 }
```

- fd: ファイルディスクリプタ
- buf: バイトを読み込むバッファ
- len: バイトを読み込む長さ

dta-dataleak – post_read_hook

Listing 11-13: dta-dataleak.cpp (post_read_hook)

```
1 uint8_t color = fd2color[fd];
2 if(color) {
3     tagmap_setn((uintptr_t)buf, len, color);
4 } else {
5     tagmap_clrn((uintptr_t)buf, len);
6 }
```

- fd にテイントカラーが割り当てられているとき
  → tagmap_setn で読み込んだバイトすべてをテイントする
- fd にテイントカラーが割り当てられていないとき（このファイルは共有ライブラリなど追跡する必要のないファイル）
  → tagmap_clrn でテイントを削除する

Contents – 11.4

dta-dataleak – pre_socketcall_hook

Listing 11-14: dta-dataleak.cpp

```
1 static void pre_socketcall_hook (syscall_ctx_t *ctx);
```

- socketcall システムコールの前に呼ばれる
- ネットワークに送信する前にデータ漏洩がないかチェックする

dta-dataleak – pre_socketcall_hook

Listing 11-14: dta-dataleak.cpp (pre_socketcall_hook)

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

- socketcall は 2つの引数を持つ
  - call: send や sendto など、どのソケット関数かを表す
  - arg: 実際のシステムコールに渡される引数のブロック

dta-dataleak – pre_socketcall_hook

Listing 11-14: dta-dataleak.cpp (pre_socketcall_hook)

```
1 switch(call) {
2 case SYS_SEND:
3 case SYS_SENDTO:
4     /* テイントの検査 */
5     break;
6 default:
7     break;
```

・ネットワーク送信機能の send, sendto のときのみテイントを調べる

dta-dataleak – pre_socketcall_hook

Listing 11-14: dta-dataleak.cpp (pre_socketcall_hook)

```
1 fd  =    (int)args[0];
2 buf =  (void*)args[1];
3 len = (size_t)args[2];
```

- fd: ソケットファイルディスクリプタ
- buf: 送信するバッファ
- len: 送信するバイトの長さ

dta-dataleak – pre_socketcall_hook

Listing 11-14: dta-dataleak.cpp (pre_socketcall_hook)

```
1 start = (uintptr_t)buf;
2 end   = (uintptr_t)buf+len;
3 for(addr = start; addr <= end; addr++) {
4     tag = tagmap_getb(addr);
5     if(tag != 0) alert(addr, tag);
6 }
```

- 送信するバッファのすべてのバイトについて tagmap_getb でテイント状態を取得する
- バイトがテイントされている (tag != 0) ときは、alert 関数でアラートを出力しプログラムを終了する

Contents – 11.4

dataleak-test-xor – main

Listing 11-15: dataleak-test-xor.c (main)

```
1 int sockfd = open_socket("localhost", "9999");
2
3 recvfrom(sockfd, buf1, sizeof(buf1), 0, (struct sockaddr*)&addr, &addrlen);
4
5 size_t fcount = split_filenames(buf1, filenames, 10);
```

- ソケットを開く
- recvfrom でファイル名のリストを受け取る
- split_filename でリストをひとつひとつのファイル名に分割する
  (fcount はファイルの個数)

dataleak-test-xor – main

Listing 11-15: dataleak-test-xor.c (main)

```
for(i = 0; i < fcount; i++) {
    fp[i] = fopen(filenames[i], "r");
}

i = rand() % fcount;
do { j = rand() % fcount; } while(j == i);
```

・要求されたすべてのファイルを開く

・これらのファイルのうちランダムに2つを選ぶ

dataleak-test-xor – main

Listing 11-15: dataleak-test-xor.c (main)

```
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);
}
```

・ランダムに選択された2つのファイルをそれぞれ一行ずつ読み取る

・XOR でそれぞれの行を結合させる
→ テイントカラーの結合をテストすることができる

・XOR で結合させた行をネットワークに送信する