CS 453/698: Software and Systems Security

Module: Other Common Vulnerability Types

Lecture: Side-channels

Meng Xu *(University of Waterloo)*

Spring 2025

Outline

How to steal sensitive information?

How to steal sensitive information?

- Install a malware (spyware) on the victim's computing device
  - e.g., screen hijacking, drive-by downloads

How to steal sensitive information?

- Install a malware (spyware) on the victim's computing device
  - e.g., screen hijacking, drive-by downloads
- Exploit a vulnerability in victim's software
  - e.g., heartbleed, log4j, etc.

How to steal sensitive information?

- Install a malware (spyware) on the victim's computing device
  - e.g., screen hijacking, drive-by downloads
- Exploit a vulnerability in victim's software
  - e.g., heartbleed, log4j, etc.
- Compromised operating system, hypervisor, or hardware
  - e.g., key logger, buggy virtualization layer, etc.

How to steal sensitive information?

- Install a malware (spyware) on the victim's computing device
  - e.g., screen hijacking, drive-by downloads
- Exploit a vulnerability in victim's software
  - e.g., heartbleed, log4j, etc.
- Compromised operating system, hypervisor, or hardware
  - e.g., key logger, buggy virtualization layer, etc.
- Side channels
  - e.g., timing, bandwidth, power, etc.

Locard's exchange principle

Locard’s exchange principle

In forensic science, **Locard’s principle** holds that: the perpetrator of a crime will bring something into the crime scene and leave with something from it, and that both can be used as forensic evidence.

$\longrightarrow$ “Every contact leaves a trace”

Locard's exchange principle

In forensic science, **Locard's principle** holds that: the perpetrator of a crime will bring something into the crime scene and leave with something from it, and that both can be used as forensic evidence.

$\longrightarrow$ "Every contact leaves a trace"

Wherever he steps, whatever he touches, whatever he leaves, even unconsciously, will serve as a silent witness against him. Not only his fingerprints or his footprints, but his hair, the fibres from his clothes, the glass he breaks, the tool mark he leaves, the paint he scratches, the blood or semen he deposits or collects. All of these and more, bear mute witness against him. This is evidence that does not forget.

— Paul L. Kirk

Locard's exchange principle (in code execution)

In forensic science, **Locard's principle** holds that: the ~~perpetrator of a crime~~ execution of code will bring something into the ~~crime scene~~ hosting platform and leave with something from it, and that both can be used as ~~forensic evidence~~ side channels.

$\longrightarrow$ *"Every contact leaves a trace"*

Locard's exchange principle (in code execution)

In forensic science, **Locard's principle** holds that: the ~~perpetrator of a crime~~ execution of code will bring something into the ~~crime scene~~ hosting platform and leave with something from it, and that both can be used as ~~forensic evidence~~ side channels.

$\longrightarrow$ *"Every contact leaves a trace"*

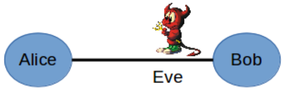
~~Wherever he steps~~ Every CPU instruction executed, ~~whatever he touches~~ every memory access, ~~whatever he leaves~~ every IO operation, even unconsciously, will serve as a silent witness against ~~him~~ the code.

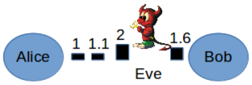
My personal story

Examples of side channels

- Bandwidth consumption
- Reflections
- Cache-timing channels

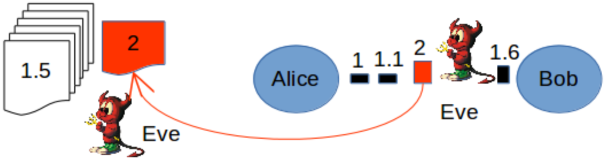
Bandwidth consumption: scenario





- Eve observes communication going via Alice's Router
- Alice accesses health forum via encrypted connection
- Eve knows that Alice connects to health forum
- But cannot decrypt downloaded content

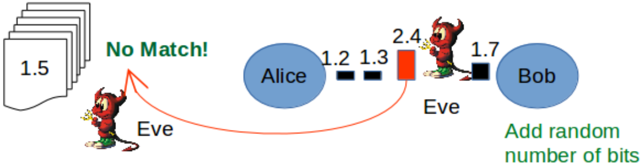
Bandwidth consumption: attack



- Eve determines size of all pages on health forum
- Eve measures size of Alice's downloaded pages
- Likely: Eve can uniquely map download to page
- This attack is called *webpage fingerprinting*
  - or *website fingerprinting*, when targeting landing pages

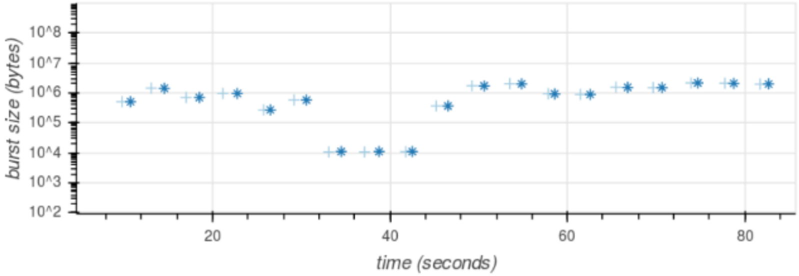
Bandwidth consumption: defense

- Pad all pages to common size (inflexible + inefficient ☹)
- Dynamic personalized websites
- (Finally a benefit of targeted advertisement)



Bandwidth consumption: another example

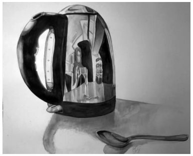
- Re-identification of Netflix video streaming
- Burst sizes of a streamed scene of “Reservoir Dogs”
  - Very similar, even when watched over different networks



Schuster et al., USENIX SEC ’17

Reflections: scenario

- Alice types her password on a device in a public place
- Alice hides her screen
- But there is a reflecting surface close by



Reflections: attack

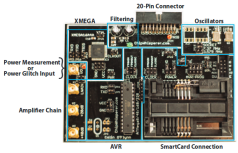
- Eve uses a camera and a telescope
- Off-the-shelf: less than CA$2,000
- Photograph reflection of screen through telescope
- Reconstruct original image
- Distance: 10–30 m
- Depends on equipment and type of reflecting surface

Reflections: defense

Other potential attack vectors

- Timing computations
- Electromagnetic emission
- Sound emissions

- Power consumption
- Differential power analysis
- Differential fault analysis



Outline

Example 1: matrix multiplication

```c
int *matrix_multiply(
    /* __secret__ */ int *A, size_t nrow_A, size_t ncol_A,
    /* __secret__ */ int *B, size_t nrow_B, size_t ncol_B
) {
    assert(ncol_A == nrow_B);
    int *C = malloc(nrow_A * ncol_B * sizeof(int));

    for(int i = 0; i < nrow_A; i++) {
        for(int j = 0; j < ncol_B; j++) {
            for(int k = 0; k < A_ncol; k++) {
                C[i * ncol_B + j] += A[i * ncol_A + k] * B[k * ncol_B + j];
            }
        }
    }
    return C;
}
```

Q: Is the above function constant-time (w.r.t secret input)?

Example 1: matrix multiplication

```c
int *matrix_multiply(
    /* __secret__ */ int *A, size_t nrow_A, size_t ncol_A,
    /* __secret__ */ int *B, size_t nrow_B, size_t ncol_B
) {
    assert(ncol_A == nrow_B);
    int *C = malloc(nrow_A * ncol_B * sizeof(int));

    for(int i = 0; i < nrow_A; i++) {
        for(int j = 0; j < ncol_B; j++) {
            for(int k = 0; k < A_ncol; k++) {
                C[i * ncol_B + j] += A[i * ncol_A + k] * B[k * ncol_B + j];
            }
        }
    }
    return C;
}
```

Q: Is the above function constant-time (w.r.t secret input)?

A: Yes

Example 1: find max

```c
int find_max(/* __secret__ */ int *arr, int n) {
    int max_val = INT_MINIMUM;
    for (int i = 0; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}
```

Q: Is the above function constant-time (w.r.t secret input)?

Example 1: find max (patched)

```c
int find_max(/* __secret__ */ int *arr, int n) {
    int max_val = INT_MINIMUM;
    for (int i = 0; i < n; i++) {
        int predicate = arr[i] > max_val;
        max_val = (predicate * arr[i])
                | (!predicate * max_val);
    }
    return max_val;
}
```

Example 2: get element

```
int get_element(
    int *arr, int size, /* __secret__ */ int index
) {
    int element = arr[index];
    return element;
}
```

Q: Is the above function constant-time (w.r.t secret input)?

Example 2: get element (patched)

```c
int get_element(
    int *arr, int size, /* __secret__ */ int index
) {
    int element = 0;
    for (int i = 0; i < size; i++) {
        int value = arr[i];
        element = select(i == index, value, element);
    }
    return element;
}
```

Example 3: constant-time instructions

```c
void foo(double x) {
    double z, y = 1.0;
    for (long i = 0; i < 100000000; i++) {
        z = y * x;
    }
}
```

Q: Which of the following execution is faster?

1. `foo(1.0)`
2. `foo(1.0e-323)`
3. they are the same

Example 3: constant-time instructions

```c
void foo(double x) {
    double z, y = 1.0;
    for (long i = 0; i < 100000000; i++) {
        z = y * x;
    }
}
```

Q: Which of the following execution is faster?

1. `foo(1.0)`
2. `foo(1.0e-323)`
3. they are the same

A: `foo(1.0)`

Rules of thumb for constant-time programming

- Avoid variable-time instructions
- If-statements on secrets are unsafe
- Memory accesses indexed by secrets are unsafe
- There are tools to help but most constant-time code is still written by hand

Constant-trace programming

A stronger version of constant-time programming: the same execution trace for all sensitive inputs.

Example 4: copy sub-array

```
void copy_subarray(
    /* secret */ char *out, /* secret */ char *in,
    uint32_t arr_len,
    uint32_t sub_len,
    /* secret */ uint32_t l_idx
) {
    uint32_t i, j;
    for(i = 0, j = 0; i < arr_len; i++) {
        if (i >= l_idx && i < l_idx + sub_len) {
            out[j] = in[i];
            j++;
        }
    }
}
```

Q: How to make the above function constant-trace?

Example 4a: copy sub-array (fix-1)

```c
// returns 0xffffffff if a < b, 0x0 otherwise
uint32_t ct_lt(uint32_t a, uint32_t b) {
    uint32_t c = a ^ ((a ^ b) | ((a - b) ^ b));
    return (0 - ( c >> (sizeof(c) * 8 - 1))) ;
}

void copy_subarray(
    /* secret */ char *out, /* secret */ char *in,
    uint32_t arr_len,
    uint32_t sub_len,
    /* secret */ uint32_t l_idx
) {
    uint32_t i, j, in_range;
    for(i = 0; i < sub_len; i++) {
        out[i] = 0;
    }
    for(i = 0, j < 0; i < arr_len; i++) {
        in_range = 0;
        in_range |= ~ct_lt(i, l_idx);
        in_range &= ct_lt(i, l_idx + sub_len);
        out[j] |= in[i] & in_range;
        j = j + (in_range & 1);
    }
}
```

Example 4b: copy sub-array (fix-2)

```c
// returns 0xffffffff if a == b, 0x0 otherwise
uint32_t ct_lt(uint32_t a, uint32_t b) {
    uint32_t c = a ^ b;
    uint32_t d = ~c & (c - 1);
    return (0 - (d >> (sizeof(d) * 8 - 1)));
}

void copy_subarray(
    /* secret */ char *out, /* secret */ char *in,
    uint32_t arr_len,
    uint32_t sub_len,
    /* secret */ uint32_t l_idx
) {
    uint32_t i, j, in_range;
    for(i = 0; i < sub_len; i++) {
        out[i] = 0;
    }
    for(i = 0; i < sub_len; i++) {
        for(j = 0; j < arr_len; j++) {
            out[j] |= in[i] & ct_eq(l_idx + j, i);
        }
    }
}
```

Reference

For more information on constant-time programming, start with the following paper: Verifying Constant-Time Implementations

$\langle$ **End** $\rangle$