Module 2: Program Security (Defenses)

entropy / moving-target defense

Meng Xu (University of Waterloo)

Spring 2023
Outline

1. Introduction
2. Stack canary
3. Randomizing memory addresses
4. Entropies in heap allocators
5. Security through diversity
Why entropy in security?

Nondeterminism is useful in software security when

- it has no impact on the intended finite state machine BUT
- limits attackers’ abilities of programming the weird machine.
Why entropy in security?

Nondeterminism is useful in software security when
- it has no impact on the intended finite state machine BUT
- limits attackers’ abilities of programming the weird machine.

In this slide deck: we will examine some standard / deployed practices of safely introducing nondeterminism to boost system and software security.
Choosing pills, a lot of pills

**Figure:** Red pill vs Blue pill. Credits / Trademark: The Matrix Movie
**Outline**

1. Introduction
2. Stack canary
3. Randomizing memory addresses
4. Entropies in heap allocators
5. Security through diversity
Recap: stack overflow

```c
int main() {
    char buf[16];
    scanf("%s", buf);
}
```
Solution 1: program analysis

```c
int main() {
    char buf[16];
    scanf("%s", buf);
}
```

Diagram:

```
<table>
<thead>
<tr>
<th>Frame Pointer</th>
<th>Return Address</th>
<th>Address of &quot;%s&quot;</th>
<th>Address of buf</th>
<th>buf (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>low address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

return address

frame pointer

buf

(16 bytes)

frame pointer

return address

high address
Solution 1: program analysis

1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4 }

1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4     scanf("%15s", buf);
5 }

low address

frame pointer

return address

address of "%s"

address of buf

buf

(16 bytes)

frame pointer

return address

high address
Solution 2: exploit mitigation

```c
int main() {
    char buf[16];
    scanf("%s", buf);
    return;
}
```

On function entry, push canary value X onto stack. On function return, check canary value is still X.
Solution 2: exploit mitigation

```c
int main() {
    char buf[16];
    scanf("%s", buf);
}
```

On function entry, push canary value X onto stack. On function return, check canary value is still X.
Solution 2: exploit mitigation

On function entry, push canary value $X$ onto stack.

On function return, check canary value is still $X$.

```
int main() {
    char buf[16];
    scanf("%s", buf);
}
```
Original use of canary

**Figure:** Canaries in coal-mining. Credits / Trademark: Alamy Stock Photo
The default implementation in GCC

```c
extern uintptr_t __stack_chk_guard;

int main() {
    noreturn void __stack_chk_fail(void);

    uintptr_t canary = __stack_chk_guard;

    char buf[16];
    scanf("%s", buf);

    if ((canary = canary ^ __stack_chk_guard) != 0) {
        __stack_chk_fail();
    }
}
```
The default implementation in GCC

```c
extern uintptr_t __stack_chk_guard;

noreturn void __stack_chk_fail(void);

int main() {
    uintptr_t canary = __stack_chk_guard;

    char buf[16];
    scanf("%s", buf);

    if ((canary = canary ^ __stack_chk_guard) != 0) {
        __stack_chk_fail();
    }
}
```

- The `__stack_chk_guard` and `__stack_chk_fail` symbols are normally supplied by a GCC library called libssp.
- You also have the option of specifying your own value for stack canaries.
### Design choices of stack canaries

<table>
<thead>
<tr>
<th>Which value should we use as canary?</th>
<th>deterministic?</th>
<th>secret?</th>
<th>random?</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the granularity of the canary invocation?</td>
<td>per function?</td>
<td>per execution?</td>
<td></td>
</tr>
<tr>
<td>When to do the integrity check?</td>
<td>on function return?</td>
<td>is that enough?</td>
<td></td>
</tr>
<tr>
<td>How much randomness is needed?</td>
<td>1 byte?</td>
<td>8 bytes?</td>
<td>64 bytes?</td>
</tr>
</tbody>
</table>
Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?

- What is the granularity of the canary invocation?
  - per function? per execution?
Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?

- What is the granularity of the canary invocation?
  - per function? per execution?

- When to do the integrity check?
  - on function return? is that enough?
Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?

- What is the granularity of the canary invocation?
  - per function? per execution?

- When to do the integrity check?
  - on function return? is that enough?

- How much randomness is needed?
  - 1 byte? 8 bytes? 64 bytes?
Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value
Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value

- Limited protection for frame pointer and return address only
  - other stack variables are not protected
Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value

- Limited protection for frame pointer and return address only
  - other stack variables are not protected

- Unable to defend against arbitrary writes
  - i.e., non-continuous overrides
1. Introduction

2. Stack canary

3. Randomizing memory addresses

4. Entropies in heap allocators

5. Security through diversity
1 int main() {
2     char buf[1024];
3     scanf("%s", buf);
4 }

\textit{low address}

\begin{itemize}
\item frame pointer
\item return address
\item address of "%s"
\item address of buf
\item buf (1024 bytes)
\item canary
\item frame pointer
\item return address
\end{itemize}

\textit{high address}
Introduction

Back to the example

```c
int main() {
    char buf[1024];
    scanf("%s", buf);
}
```

Meaningful values for return address:

- Shellcode (stack)
- `system()` in libc

```
low address

frame pointer
return address
address of "%s"
address of buf
buf (1024 bytes)
...canary
frame pointer
return address
...high address
```
1 int main() {
2     char buf[1024];
3     scanf("%s", buf);
4 }

Meaningful values for return address:
- Shellcode (stack)
- system() in libc
Randomize the addresses

**ASLR** — Address Space Layout Randomization, is a system-level protection that **randomly** arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

**PIE** — Position Independent Executable, is a body of machine code that executes properly **regardless of its absolute address**. This is also known as position-independent code (PIC).
Base case: static program

- Fixed address
  - .text
  - .bss + .data
  - Heap

- Fixed address
  - Stack
  - Env

- Fixed address
  - low address

- Fixed address
  - high address

Env

Stack

Heap

.text

.bss + .data

low address

high address
Static program + shared libraries + ASLR

- Fixed address
- Randomized address
- Stack
- Env

- Heap
- .text
- .bss + .data

- libc.so
- ld.so

- low address
- high address
Static program + shared libraries + ASLR + PIE

- Randomized address
- .text
- .bss + .data
- Heap
- .text
- .bss + .data
- ld.so
- libc.so
- Stack
- Env
- low address
- high address
Paranoid randomization

Figure: Different level of randomization proposed by the ASLR-NG project
Limitations of ASLR + PIE

- Limited entropy
  - visualized by the ASLR-NG project
Limitations of ASLR + PIE

- Limited entropy
  - visualized by the ASLR-NG project

- Memory layout inheritance
  - Child processes inherit/share the memory layout of the parent.
Outline

1. Introduction
2. Stack canary
3. Randomizing memory addresses
4. Entropies in heap allocators
5. Security through diversity
Motivation for secure heap allocators

Memory errors are equally (if not more) likely to happen on heap objects which can cause all sorts of unexpected behaviors.
A heap buffer overflow case

```c
struct dispatcher {
    uint64_t counter;
    int (*action)(uint64_t counter, char *data);
}

int main() {
    char *p1 = malloc(16);
    char *p2 = malloc(sizeof(struct dispatcher));
    p2->counter = 0;
    p2->action = /* some valid function */;

    scanf("%s", p1);
    int result = p2->action(p2->counter, p1);

    free(p1);
    free(p2);
    return result;
}
```
A heap use-after-free case

```c
struct dispatcher {
    uint64_t counter;
    int (*action)(uint64_t counter, char *data);
};

char *p1;

void main() {
    p1 = malloc(16);
    pthread_create(/* ... */ , thread_1);
    pthread_create(/* ... */ , thread_2);
    /* wait for thread termination */
}

void thread_1() {
    scanf("%15s", p1);
    /* ... compromised here ... */
    /* use-after-free */
    free(p1);
    ((struct dispatcher *)p1)
        ->action = /* bad function */;
}

void thread_2() {
    char *p2 = malloc(
        sizeof(struct dispatcher));
    p2->counter = 0;
    p2->action = /* good function */;
    p2->action(p2->counter, p1);
    free(p2);
}
```
Secure heap allocators

These exploits have implicit assumptions on the layout of the heap, which can be invalidated by a secure heap allocator.
Basic allocator example

Initial state:   

p1 = malloc(16);  

p2 = malloc(sizeof(..));  

free(p1);  

p3 = malloc(sizeof(..));

Each square is a 4-byte box
Allocation + random placement

Initial state:

```
p1 = malloc(16);
p2 = malloc(sizeof( .. ));
free(p1);
p3 = malloc(sizeof( .. ));
```

0 Each square is a 4-byte box
Allocator + random placement + canary

Initial state:

\[
\begin{align*}
p1 &= \text{malloc}(16); \\
p2 &= \text{malloc}(\text{sizeof}(..)); \\
\text{free}(p1); \\
p3 &= \text{malloc}(\text{sizeof}(..));
\end{align*}
\]

\(^0\) Each square is a 4-byte box
Outline

1. Introduction
2. Stack canary
3. Randomizing memory addresses
4. Entropies in heap allocators
5. Security through diversity
In biology, maintaining high genetic diversity allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.
Intuition: gene/DNA diversity

In biology, maintaining high genetic diversity allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.

Similarly, we expect software diversity to protect software systems (especially critical systems) from deadly viruses and attacks while also serving as an early signal of being attacked.
Core architecture (under attack)

Input dispatching

Instance 0  Instance 1  \cdots  Instance N

Synchronization & output aggregation
Challenges of applying diversity-based defenses

- Source of diversity
- Synchronization of diversified instances
Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation
Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation

- N-version programming
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams
Source of diversity

- **Compiler/loader-assisted diversity**
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation

- **N-version programming**
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams

- **Platform diversity**
  - e.g., different libc implementations (glibc vs musl libc)
  - e.g., Adobe Reader on MacOS and Windows
  - e.g., Server programs on Intel and ARM CPUs
Mode of synchronization

- Online mode (via rendezvous points)
- Offline mode (via record-and-replay)

The key is to synchronize all sources of nondeterminism.
〈 End 〉