SPAM
Stateless Permutation of Application Memory
With LLVM

Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Simha Sethumadhavan
Columbia University
10/8/20
About us

*Miguel A. Arroyo*
5th year PhD Candidate

@miguelaarroyo12
https://miguel.arroyo.me

*Mohamed Tarek*
4th year PhD Candidate

@M_TarekIbnZiad
https://cs.columbia.edu/~mtarek
Memory Safety is a serious problem!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google's analysis of the novel hacking operation.
Memory Safety is a serious problem!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
Memory Safety is a serious problem!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.

-- The New York Times

WhatsApp Rushes to Fix Security Flaw Exposed in Hacking of Lawyer’s Phone

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
It’s easy to make mistakes
Prevalence of Memory Safety Vulns

Microsoft Product CVEs

Google Chrome Bug Report 2015-2020

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019

Source: https://www.chromium.org/Home/chromium-security/memory-safety
ATTACKERS

MEMORY SAFETY
Attackers Prefer Memory Safety Vulns

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019
À la carte solutions with additive overheads

<table>
<thead>
<tr>
<th>Memory Safety Menu</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra-Object Overflow</td>
<td>$$$</td>
</tr>
<tr>
<td>Inter-Object Overflow</td>
<td>$$</td>
</tr>
<tr>
<td>Buffer-Overread</td>
<td>$</td>
</tr>
<tr>
<td>Control-Flow Hijack</td>
<td>$</td>
</tr>
<tr>
<td>Use-after-free</td>
<td>$$</td>
</tr>
<tr>
<td>Type Confusion</td>
<td>$$$</td>
</tr>
<tr>
<td>Uninitialized Reads</td>
<td>$$</td>
</tr>
</tbody>
</table>
No common solution to all problems

Spectre  Meltdown  RowHammer  RamBleed  ColdBoot
Overview
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));

main.c
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));

main.c
1

Object Allocation
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;
Object Allocation
Generating Permutations

1. Request memory from allocator.
Object Allocation

Generating Permutations

```c
typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;
```

1. Request memory from allocator.
2. Use address as key for permutation.
Object Allocation

Generating Permutations

1. Request memory from allocator.
2. Use address as key for permutation.
3. Write to memory in permuted order.
Object Allocation

Generating Permutations

1. Request memory from allocator.
2. Use address as key for permutation.
3. Write to memory in permuted order.
Overview

Struct Definition

typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

Virtual Address (VA)

Object Allocation

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));

Object Deallocation & Reuse

main.c
Object Deallocation & Reuse
Object Deallocation & Reuse
Object Deallocation & Reuse

A1

0x00004000

free
Object Deallocation & Reuse

A1: 0x00004000

A3: 0x00004000

Time
Object Deallocation & Reuse

Same address means same layout!
Object Deallocation & Reuse

A1

0x00004000

free

A3

0xCAFE4000

Alias Number

Time
Object Deallocation & Reuse

A1 \(0x00004000\)

free

A3 \(0xCAFE4000\)
Object Deallocation & Reuse

Different Layouts!

A1 → 0x00004000 → A3

0xCAFE4000

Time
Overview

Struct Definition

typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

A_t *A1 = malloc(sizeof(A_t));
A_t *A2 = malloc(sizeof(A_t));
free(A1);
A_t *A3 = malloc(sizeof(A_t));

Object Allocation

Object Deallocation & Reuse

Virtual Address (VA)

Multi-Dimensional Objects

typedef struct {
    char a;
    double b;
    A_t_c *c_ptr;
    void (*fp)();
} A_t;

typedef struct {
    char c[3];
} A_t_c;

main.c
Multi-Dimensional Objects
Multi-Dimensional Objects

Buf2Ptr Transformation

typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

typedef struct {
    char c[3];
} A_t_c;

typedef struct {
    char a;
    double b;
    A_t_c *c_ptr;
    void (*fp)();
} A_t;
Multi-Dimensional Objects

Buf2Ptr Transformation

typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

typedef struct {
    char c[3];
} A_t_c;

typedef struct {
    char a;
    double b;
    A_t_c *c_ptr;
    void (*fp)();
} A_t;
Multi-Dimensional Objects

Allocation & Permutation

typedef struct {
    char a;
    double b;
    A_t_c *c_ptr;
    void (*fp)();
} A_t_c;

typedef struct {
    char c[3];
} A_t_c;

Virtual Address (VA)

A1 → 0xC0C040FF

A1_c → 0xC0C040FF
Implementation
Buf2Ptr: Source-to-Source Transformation

(a) Original

```c
struct Foo {
    char buf[10];
};
```

(b) Transformed

```c
// Promoted Type
struct Foo_buf {
    char buf[10];
};
struct Foo {
    struct Foo_buf *p_buf;
};
```
Buf2Ptr: Source-to-Source Transformation

(a) Original

```c
struct Foo {
    char buf[10];
};

struct Foo *f = malloc(sizeof(struct Foo));
f->buf[7] = 'A';
free(f);
```

(b) Transformed

```c
// Promoted Type
struct Foo_buf {
    char buf[10];
};
struct Foo {
    struct Foo_buf *p_buf;
};
```

```c
// Promoted Allocations
struct Foo *f = malloc(sizeof(struct Foo));
f->p_buf = malloc(sizeof(struct Foo_buf));
free(f);
```
Buf2Ptr: Source-to-Source Transformation

(a) Original

```c
struct Foo {
    char buf[10];
};

struct Foo *f = malloc(sizeof(struct Foo));
f->buf[7] = 'A';
free(f);
```

(b) Transformed

```c
// Promoted Type
struct Foo_buf {
    char buf[10];
};
struct Foo {
    struct Foo_buf *p_buf;
};

struct Foo *f = malloc(sizeof(struct Foo));
f->p_buf = malloc(sizeof(struct Foo_buf));
free(f);
free(f->p_buf);
```
Buf2Ptr: Source-to-Source Transformation

(a) Original

```
struct Foo {
    char buf[10];
};

struct Foo *f = malloc(sizeof(struct Foo));

f->buf[7] = 'A';

free(f);
```

(b) Transformed

```
// Promoted Type

struct Foo {
    char buf[10];
};

struct Foo {
    struct Foo_buf *p_buf;
};

// Promoted Allocations

struct Foo *f = malloc(sizeof(struct Foo));

f->p_buf = malloc(sizeof(struct Foo_buf));

// Promoted Usages


// Promoted Deallocations

free(f->p_buf);

free(f);
```
Framework

Clang

C Code

C Code

Src-to-Src Transformation

LLVM

IR

Inst. Pass

Object Files

Runtime Library

Linker

SPAM

Hardened Binary

46
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
Instrumentation & Runtime

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
```

Baseline Compilation

*Flags: -00*
Instrumentation & Runtime

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
**Global Support**

```c
void RegisterGlobal(void *Ptr)
```

For `.data` section hook into `.ctor` to permute on program load.

**Stack Support**

```c
void *RegisterStack(void *Ptr)
```

For variables passed by OS (e.g. `argv`) hook into `main` to permute on start.
Why SPAM?
SPAM Benefits

Spatial Memory Safety

- Every object instance (allocation) is permuted independently.
- Overflows within an object (intra) are transformed.
SPAM Benefits

Spatial Memory Safety

Byte-granular protection!
SPAM Benefits

- Spatial Memory Safety
- Temporal Memory Safety

Byte-granular protection!

- Alias number introduces additional entropy when an address is reused.
SPAM Benefits

Spatial Memory Safety

Temporal Memory Safety

Byte-granular protection!

No need for quarantine!
SPAM Benefits

- Spatial Memory Safety
  - Byte-granular protection!
- Temporal Memory Safety
  - No need for quarantine!
- Side-Channel & Fault Resilience
  - Data is permuted across the memory hierarchy.
SPAM Benefits

- Spatial Memory Safety
- Temporal Memory Safety
- Side-Channel & Fault Resilience

- Byte-granular protection!
- No need for quarantine!
- Unified protection for SW & HW Vulns!
SPAM Benefits

- Spatial Memory Safety
  - Byte-granular protection!

- Temporal Memory Safety
  - No need for quarantine!

- Side-Channel & Fault Resilience
  - Unified protection for SW & HW Vulns!

- Stateless
  - No opportunity for manipulation by an attacker.
SPAM Benefits

- **Spatial Memory Safety**: Byte-granular protection!
- **Temporal Memory Safety**: No need for quarantine!
- **Side-Channel & Fault Resilience**: Unified protection for SW & HW Vulns!
- **Stateless**: Support for multithreading!
Resilience to Common Exploits
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot reliably corrupt memory.

Buffer A
Buffer B
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot reliably corrupt memory.

Original: Buffer A Buffer B

Buffer A Buffer B
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot reliably corrupt memory.

Buffer A
Buffer B

Original:
Buffer A
Buffer B

SPAM:
Buffer A
Buffer B

Non-linear write can trigger exception!
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot reliably corrupt memory.

2. Use-after-free
   Each instance permuted independently.

Alias number provides multiple permutations.

Alias Number  Virtual Address
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot reliably corrupt memory.

2. Use-after-free
   Each instance permuted independently.

3. Speculative Attacks
   Speculative load uses a different permutation to access the permuted data.

// mispredicted branch
if (i < sizeof(a)) {
    secret = a[i];

    // secret is leaked
    val = b[64 * secret];
}

- Attacker will end up with an unpredictable value in secret due as the permutation depends on the address of a[i].
SPAM Meets Reality
SPAM Meets Reality
Compatibility with Uninstrumented Code
SPAM Meets Reality
Compatibility with Uninstrumented Code

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *p = malloc(128);
    *p = 'A';
    printf("%c\n", *p);
    return 0;
}
```

```c
int printf(const char *fmt, ...) {
    int err;

    va_list ap;
    va_start(ap, fmt);
    err = _dvprintf(fmt, ap);
    va_end(ap);

    return err;
}
```
SPAM Meets Reality
Compatibility with Uninstrumented Code
SPAM Meets Reality

Compatibility with Uninstrumented Code

Other Memory

SPAM Permuted Domain

External Unpermuted Domain
SPAM Meets Reality

Compatibility with Uninstrumented Code
SPAM Meets Reality
Compatibility with Uninstrumented Code

```c
void *Unpermute(void *Ptr)
```

Other Memory

SPAM Permuted Domain

External Unpermuted Domain
SPAM Meets Reality
Compatibility with Uninstrumented Code
SPAM Meets Reality
Compatibility with Uninstrumented Code
SPAM Meets Reality
Compatibility with Uninstrumented Code

`void *Permute(void *Ptr)`

Other Memory

SPAM Permutated Domain

External Unpermuted Domain
SPAM Meets Reality
Compatibility with Uninstrumented Code

SPAM Permuted Domain

External Unpermuted Domain

Other Memory
SPAM Meets Reality
Compatibility with Uninstrumented Code
SPAM Meets Reality
Compatibility with Uninstrumented Code

```c
int cmp (const void * a,
         const void * b) {...}

int main() {
    ...
    qsort(b, 10, 10, cmp);
    ...
}

void qsort(void *base,
           size_t nitems,
           size_t size,
           int (*cmp)(const void *, const void*))
{...}
```

SPAM
Permuted Domain

External
Unpermuted Domain
SPAM Meets Reality

Compatibility with Uninstrumented Code

```c
int cmp (const void * a,
    const void * b) {...}

int main() {
    ...
    qsort(b, 10, 10, cmp);
    ...
}

void qsort(void *base,
    size_t nitems,
    size_t size,
    int (*cmp)(const void *, const void*))
{...}
```

SPAM Permuted Domain

External Unpermuted Domain
SPAM Meets Reality
Compatibility with Uninstrumented Code

int cmp (const void * a,
         const void * b) {...}

int main() {
...
    qsort(b, 10, 10, cmp);
...
}

void qsort(void *base,
           size_t nitems,
           size_t size,
           int (*cmp)(const void *, const void*))
{...}

SPAM
Permuted Domain

External
Unpermuted Domain
int main() {
    ...
    qsort(b, 10, 10, cmp);
    ...
}

int cmp (const void * a, const void * b) {...}

void qsort(void *base, size_t nitems,
           size_t size,
           int (*cmp)(const void *, const void*))
        {...}
SPAM Meets Reality

Hardware Support

Accelerate Permutation!
SPAM Meets Reality

Hardware Support

```assembly
%store_off = call i8* @spam_get_perm_offset(i8* %ptr, i8* %ptr)

store i8 65, i8* %store_off, align 1

%load_off = call i8* @spam_get_perm_offset(i8* %ptr, i8* %ptr)

%load = load i8, i8* %load_off, align 1
```
SPAM Meets Reality

Hardware Support

%store_off = call i8* @spam_get_perm_offset(i8* %ptr, i8* %ptr)

store i8 65, i8* %store_off, align 1

%load_off = call i8* @spam_get_perm_offset(i8* %ptr, i8* %ptr)

%load = load i8, i8* %load_off, align 1

spam_store i8 65, i8* %ptr, align 1

%load = spam_load i8, i8* %ptr, align 1

Reduce resource pressure!
Other Mitigations
Other Mitigations

• ARM MTE
  • Memory & pointers are tagged with colors.

Other Mitigations

- **ARM MTE**
  - Memory & pointers are tagged with colors.
  
  Limited set of colors.

Other Mitigations

• ARM MTE
  • Memory & pointers are tagged with colors.

Vulnerable to intra-object & type confusion.

Other Mitigations

- **ARM MTE**
  - Memory & pointers are tagged with colors.

- **Checked C**
  - Adds new pointer and array types that are bounds checked.

Other Mitigations

• **ARM MTE**
  • Memory & pointers are tagged with colors.

• **Checked C**
  • Adds new pointer and array types that are bounds checked.

Other Mitigations

No Hardware Side-Channel Resilience!
Prototype Results
Prototype Results
Average Performance Overheads

- C only subset of programs.
- Google Chrome’s Octane 2 Benchmark Suite
- Included Wolfcrypt benchmarks.

- ~2.11x overhead
- ~1.4x overhead
- ~3.15x overhead
- ~2.48x overhead
Unsupported Functionality
Unsupported Functionality

- **Inline Assembly**
  - Can be handled with lifting or (un)permute primitives.
Unsupported Functionality

• **Inline Assembly**
  • Can be handled with lifting or (un)permute primitives.

• **Variadic Functions**
  • Invoking functions with `va_list` as an argument (e.g. `vsprintf`) are unsupported.

```c
void my_printf(const char *fmt, ...)
{
    char buffer[256];

    va_list ap;
    va_start(ap, fmt);
    vprintf(buffer, fmt, ap);
    va_end(ap);
}
```
Unsafe Functionality

- **Inline Assembly**
  - Can be handled with lifting or (un)permute primitives.

- **Variadic Functions**
  - Invoking functions with `va_list` as an argument (e.g. `vsprintf`) are unsupported.

```c
void my_printf(const char *fmt, ...)
{
    char buffer[256];

    va_list ap;
    va_start(ap, fmt);
    vsprintf(buffer, fmt, ap);
    va_end(ap);
}
```
SPAM (in a nutshell)

• Unified solution to multiple software and hardware memory security issues.
SPAM (in a nutshell)

• Unified solution to multiple software and hardware memory security issues.
  ▪ Key Features
    ✓ Metadata-less: enabled by permuting based on allocation address and a salt.
**SPAM (in a nutshell)**

- Unified solution to multiple software and hardware memory security issues.
  - Key Features
    - Metadata-less: enabled by permuting based on allocation address and a salt.
    - Out-of-the-box compatibility with MT code: due to metadata-less nature.
SPAM (in a nutshell)

• Unified solution to multiple software and hardware memory security issues.
  ▪ Key Features
    ✓ Metadata-less: enabled by permuting based on allocation address and a salt.
    ✓ Out-of-the-box compatibility with MT code: due to metadata-less nature.
    ✓ Compatible with non-SPAM code: allows incremental adoption.
SPAM (in a nutshell)

- Unified solution to multiple software and hardware memory security issues.

  - Key Features
    - Metadata-less: enabled by permuting based on allocation address and a salt.
    - Out-of-the-box compatibility with MT code: due to metadata-less nature.
    - Compatible with non-SPAM code: allows incremental adoption.
    - Suitable for HW acceleration: localized changes within the pipeline.
SPAM (in a nutshell)

• Unified solution to multiple software and hardware memory security issues.
  ▪ Key Features
    ✓ Metadata-less: enabled by permuting based on allocation address and a salt.
    ✓ Out-of-the-box compatibility with MT code: due to metadata-less nature.
    ✓ Compatible with non-SPAM code: allows incremental adoption.
    ✓ Suitable for HW acceleration: localized changes within the pipeline.

Future Work

• C++ support
• HW support (including 32-bit systems)
**SPAM (in a nutshell)**

- Unified solution to multiple software and hardware memory security issues.
  - **Key Features**
    - *Metadata-less*: enabled by permuting based on allocation address and a salt.
    - *Out-of-the-box compatibility with MT code*: due to metadata-less nature.
    - *Compatible with non-SPAM code*: allows incremental adoption.
    - *Suitable for HW acceleration*: localized changes within the pipeline.

**Future Work**

- C++ support
- HW support (including 32-bit systems)

Checkout our technical report on Arxiv!