Changelog

Corrections made in this version not in first posting:

3 April 2017: Fix ROP with VTable overwrite example (slide 11) to use %rsi instead of %rdi. I somehow thought *(%rdi) was looking for a pointer to pointer when it certainly does not

last time

ASLR — random addresses

- performance/compatibility concerns

write XOR execute — no injecting machine code

- minor compatibility concerns

ROP — defeating write XOR execute

logistical notes

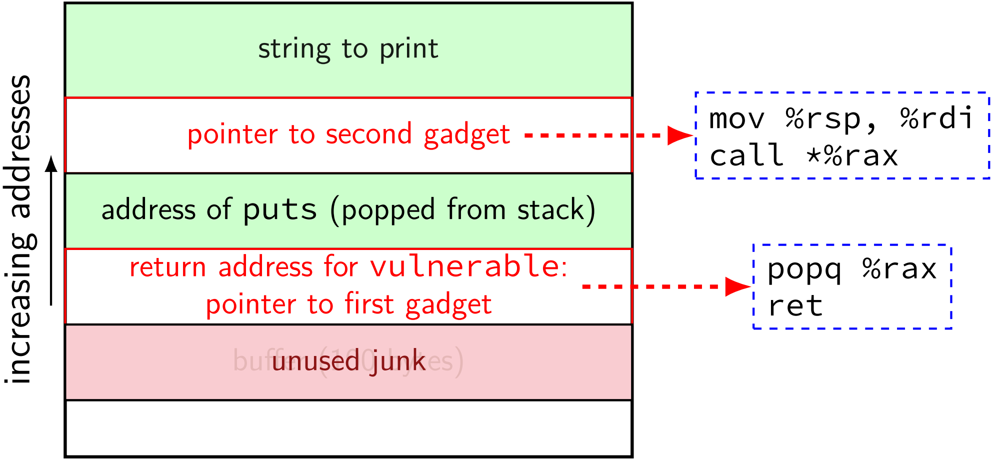
exam review — questions?

FORMAT

on the final

- likely part take-home, part in-class

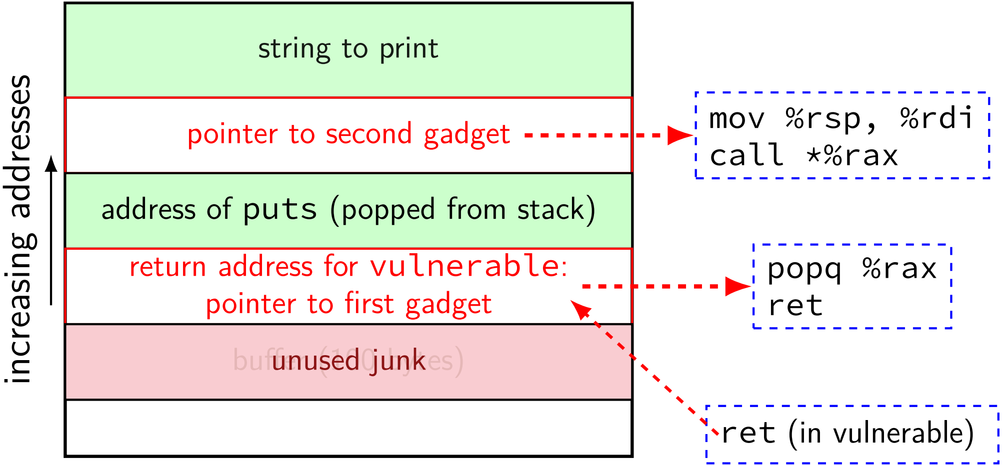
ROP chain



```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

ROP chain



string to print

pointer to second gadget

address of `puts` (popped from stack)

return address for `vulnerable`:

pointer to first gadget

buffer (100 bytes)

unused junk

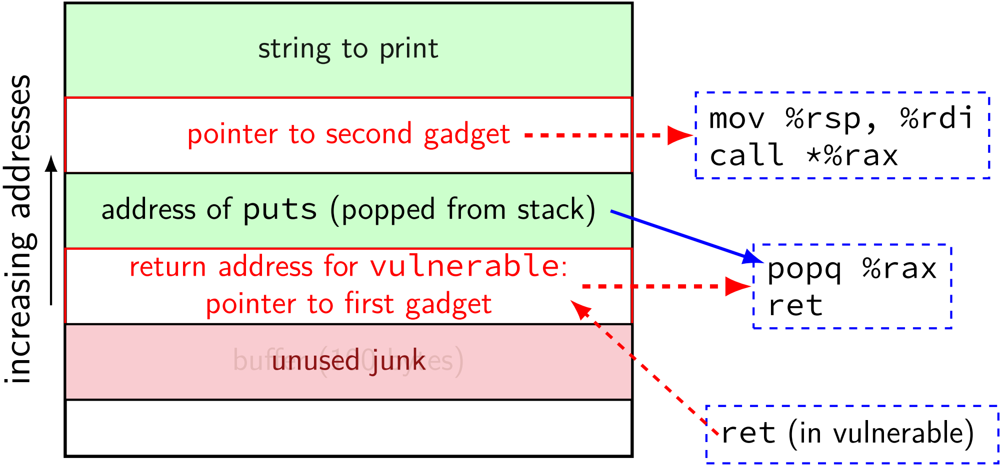
increasing addresses

```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

`ret` (in vulnerable)

ROP chain



increasing addresses

string to print

pointer to second gadget

address of `puts` (popped from stack)

return address for `vulnerable`:

pointer to first gadget

buffer (100 bytes)

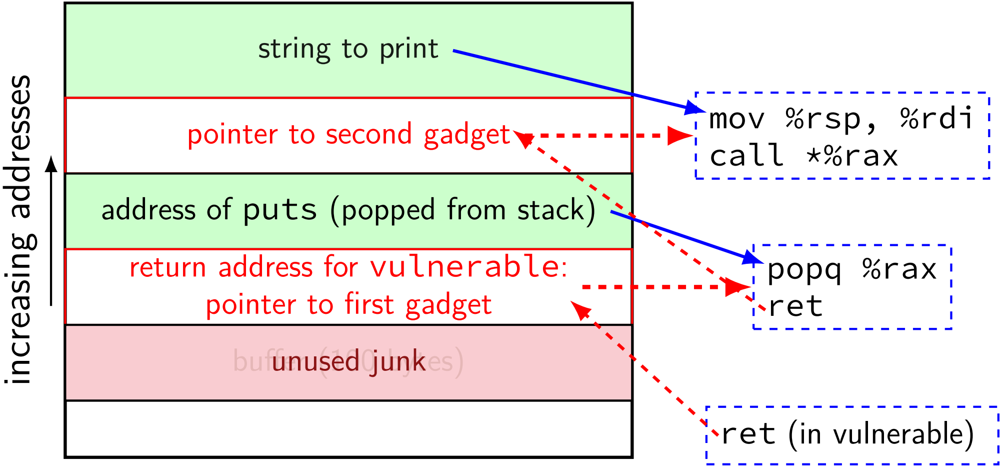
unused junk

```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

`ret` (in vulnerable)

ROP chain



increasing addresses

string to print

pointer to second gadget

address of `puts` (popped from stack)

return address for `vulnerable`:
pointer to first gadget

buffer (100 bytes)

unused junk

```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

`ret` (in vulnerable)

gadgets generally

bits of machine code that do work, then return or jump

“chain” together, by having them jump to each other

most common: find gadget ending with `ret`

    pops address of next gadget offs tack

can do pretty much anything

ROP and ASLR

find a pointer to known thing in libc (or other source of gadgets)

- e.g. information leak from use-after-free

use that to compute address of all gadgets

then address randomization doesn't matter

ROP and write XOR execute

all the code we're running is supposed to be executed

completely defeats write XOR execute

ROP and stack canaries

information disclosure reveals canary value if needed, still

full stack canaries should reduce number of gadgets

- no real returns without canary checks

…but typically only canaries if stack-allocated buffer

and return opcodes within other instructions

ROP without a stack overflow (1)

e.g. VTable overwrite

look for gadget(s) that set `%rsp`

...based on function argument registers/etc.

ROP without stack overflow (2)

example sequence:

~~`push %rdi; call *(%rdx)`~~

forgot to account for call last time

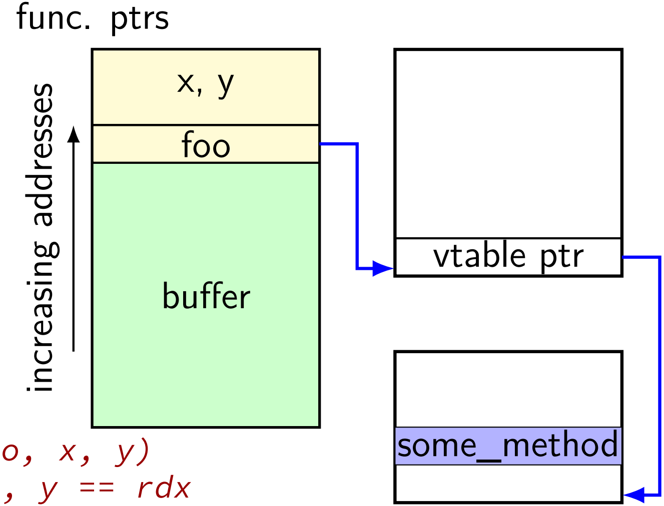
```
push %rdx; jmp *(%rsi)
pop %rsp; ret
```

set:

overwritten vtable entry = pointer to first gadget
arg 2: `%rsi` = pointer to pointer to second gadget
arg 3: `%rdx` = desired stack pointer

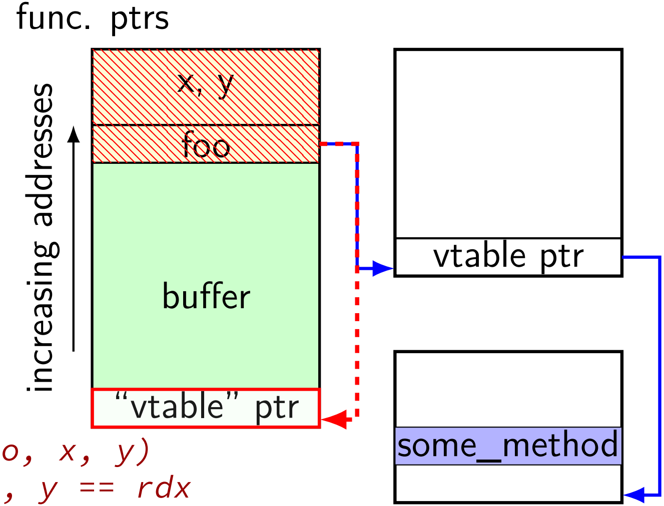
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



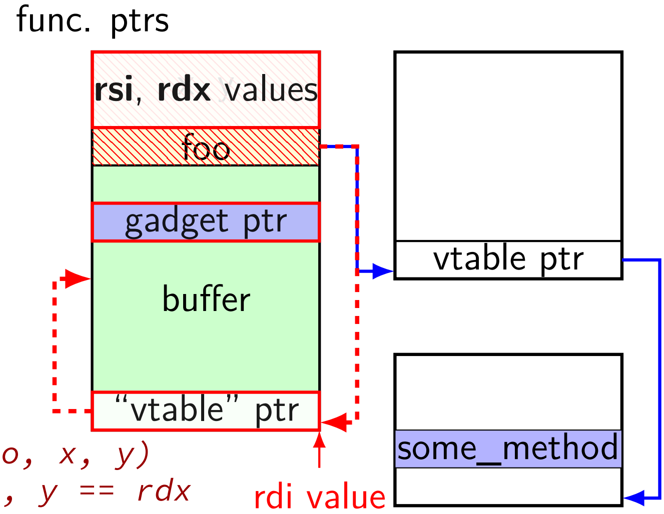
VTable overwrite with gadget

```
class Bar {
   char buffer[100];
   Foo *foo;
   int x, y;
   ...
};

void Bar::vulnerable() {
   gets(buffer);
   foo->some_method(x, y);
   // (*foo->vtable[K])(foo, x, y)
   // foo == rdi, x == rsi, y == rdx
}
```



VTable overwrite with gadget

```
class Bar {
    char buffer[100];
    Foo *foo;
    int x, y;
    ...
};

void Bar::vulnerable() {
    gets(buffer);
    foo->some_method(x, y);
    // (*foo->vtable[K])(foo, x, y)
    // foo == rdi, x == rsi, y == rdx
}
```



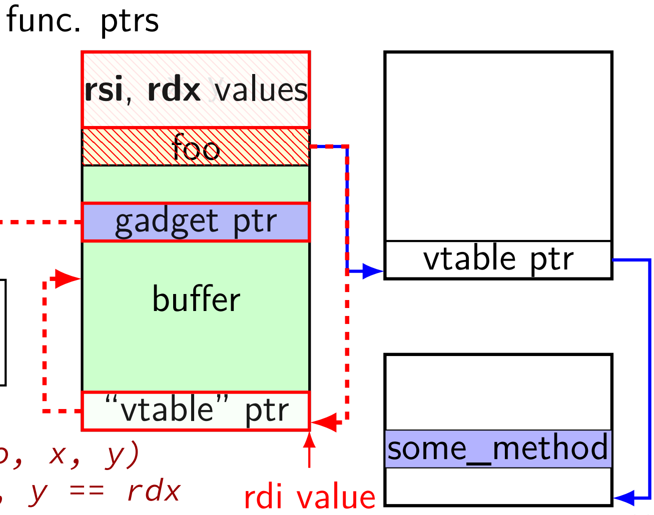
VTable overwrite with gadget

```
class Bar {
    char buffer[100];
    Foo *foo;
    int x, y;
    ...
};
```

gadget:

```
push %rdx; jmp *(%rsi)
```

```
    foo->some_method(x, y);
    // (*foo->vtable[K])(foo, x, y)
    // foo == rdi, x == rsi, y == rdx
}
```



jump-oriented programming

just look for gadgets that end in `call` or `jmp`

don't even need to set stack

harder to find than `ret`-based gadgets

- but almost always as powerful as ret-based gadgets

makes return-oriented programming mitigation hard

- can't just protect all `ret`s (in middle of instruction or not)

finding gadgets

find code segments of exectuable/library

look for opcodes of arbitrary jumps:

```
ret
jmp *register
jmp *(register)
call *register
call *(register)
```

disassemble starting a few bytes before

- invalid instruction? jump before ret? etc. — discard

sort list

programming with gadgets

can usually find gadgets to:

- pop from stack into argument register
- write register to memory location in another register
- clear a register

along with gadget for `syscall` (make OS call) — can do anything

common, reusable ROP sequences

open a command-line — what `ROPgadget` tool defaults to

make memory executable + jump

- generally: just do enough to ignore write XOR execute

often only depend on memory locations in shared library

- works across programs — e.g. many programs use the C standard library

ROP ideas

incidental *existing* snippets of code

chain together with non-constant jumps

  returns, function pointer calls, computed jumps

snippets form “language”

  usually Turing-complete

next topic: fixing real problems

we've focused on "band-aid" solutions

- detect memory corruption; then hope you can do something

first idea everyone has: just add bounds-checking!

- Java, Python do it…

adding bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

couldn't compiler add check for `len`

modern Linux: it does

added bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

```
    subq   $72, %rsp
    leaq   4(%rsp), %rdi
    movslq len, %rdx
    movq   attacker_controlled, %rsi
    movl   $42, %ecx
    call   __memcpy_chk
```

length 42 passed to `__memcpy_chk`

_FORTIFY_SOURCE

Linux C standard library + GCC features

adds automatic checking to a bunch of string/array functions

even printf (disable %n unless format string is a constant)

often enabled by default

GCC options:

- `-D_FORTIFY_SOURCE=1` — enable (backwards-compatible only)
- `-D_FORTIFY_SOURCE=2` — enable (full)
- `-U_FORTIFY_SOURCE` — disable

non-checking library functions

some C library functions make bounds checking hard:

```
strcpy(destination, source);
strcat(destination, source);
sprintf(destination, format, ...);
```

bounds-checking versions (added to library later):

```
/* might not add \0 (!) */
strncpy(destination, source, size);
// destination[size - 1] = '\0';
/* will add \0: */
strncat(destination, source, size);
snprintf(destination, size, format, ...);
```

C++ bounds checking

```
#include <vector>
...
std::vector<int> data;
data.resize(50);
// undefined behavior:
data[60] = 0;
// throws std::out_of_range exception
data.at(60) = 0;
```

language-level solutions

languages like Python don't have this problem

couldn't we do the same thing in C?

bounds-checking C

there have been many proposals to add bounds-checking to C

including implementations

brainstorm: why hasn't this happened?

easy bounds-checking

```
void vulnerable() {
    char buffer[100];
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        buffer[i] = c;
    }
}
void vulnerable_checked() {
    char buffer[100];
    int c;
    int i = 0;
    while ((c = getchar()) != EOF && c != '\n') {
        CHECK(i >= 100 || i < 0);
        buffer[i] = c;
    }
}
```

adding bounds-checking — fat pointers

```
struct MyPtr {
    char *pointer;
    char *minimum;
    char *maximum;
};
```

adding bounds checking — strcpy

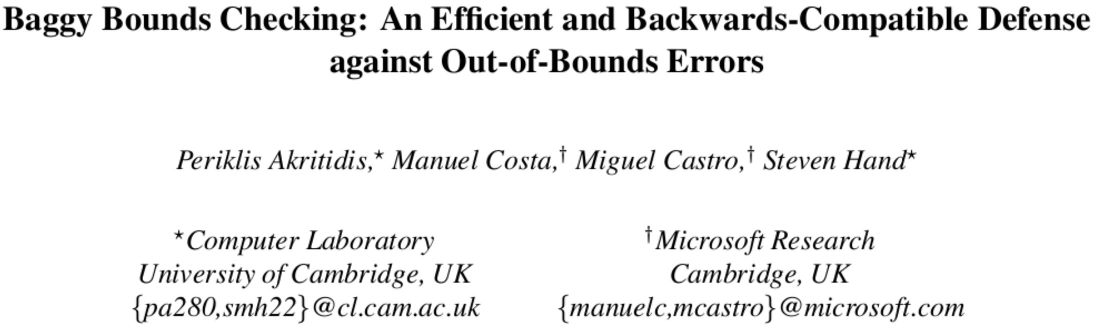
```
MyPtr strcpy(MyPtr dest, const MyPtr src) {
    int i;
    do {
        CHECK(src.pointer + i <= src.maximum);
        CHECK(src.pointer + i >= src.minimum);
        CHECK(dest.pointer + i <= dest.maximum);
        CHECK(dest.pointer + i >= dest.minimum);
        src.pointer[i] = dest.pointer[i];
        i += 1;
        CHECK(src.pointer + i <= src.maximum);
        CHECK(src.pointer + i >= src.minimum);
    } while (src.pointer[i] != '\0');
    return dest;
}
```

speed of bounds checking

two comparisons for every pointer access?

three times as much space for every pointer?

research example (2009)



Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors

Periklis Akritidis,[*] Manuel Costa,[†] Miguel Castro,[†] Steven Hand[*]

[*]*Computer Laboratory*
University of Cambridge, UK
{pa280,smh22}@cl.cam.ac.uk

[†]*Microsoft Research*
Cambridge, UK
{manuelc,mcastro}@microsoft.com

baggy bounds checking idea

giant lookup table — one entry for every 16 bytes of memory

table indicates start of object allocated here

check pointer arithmetic:

```
char p = str[i];
/* becomes: */
CHECK(START_OF[str / 16] == START_OF[&str[i] / 16]);
char p = str[i];
```

baggy bounds trick

table of pointers to starting locations would be huge

add some restrictions:

- all object sizes are powers of two
- all object starting addresses are a multiple of their size

then, table contains size info only:

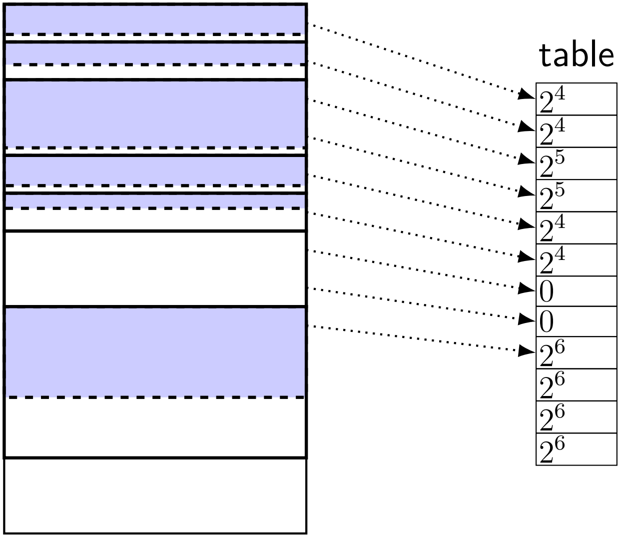
- table contains i, size is 2^i bytes:

```
char *GetStartOfObject(char *pointer) {
    return pointer & ~(1 << TABLE[pointer / 16] - 1);
    /* pointer bitwise-and 2^(table entry) - 1 */
    /* clear lower (table entry) bits  of pointer */
}
```

allocations and lookup table

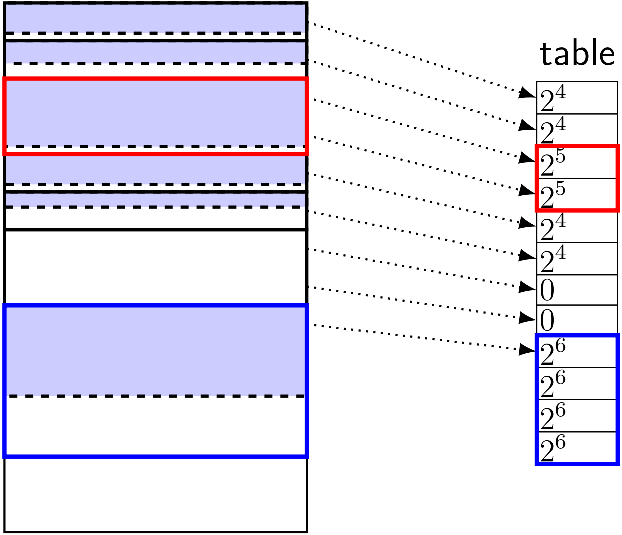
object allocated in
power-of-two 'slots'

allocations and lookup table



object allocated in power-of-two ‘slots’

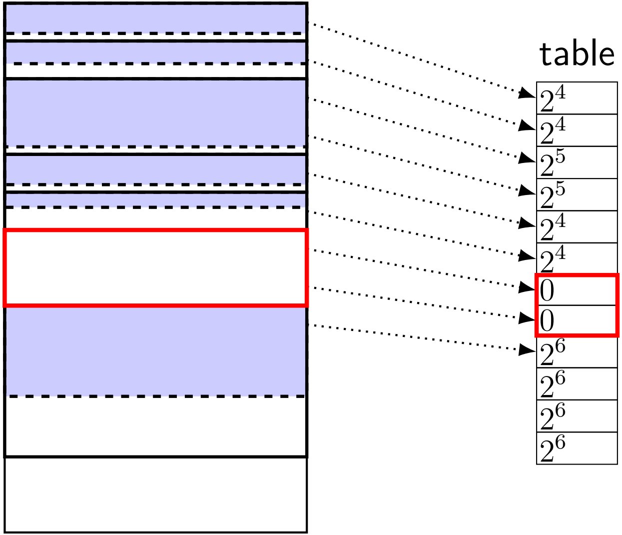
allocations and lookup table



object allocated in power-of-two ‘slots’

table stores sizes
for each 16 bytes

allocations and lookup table

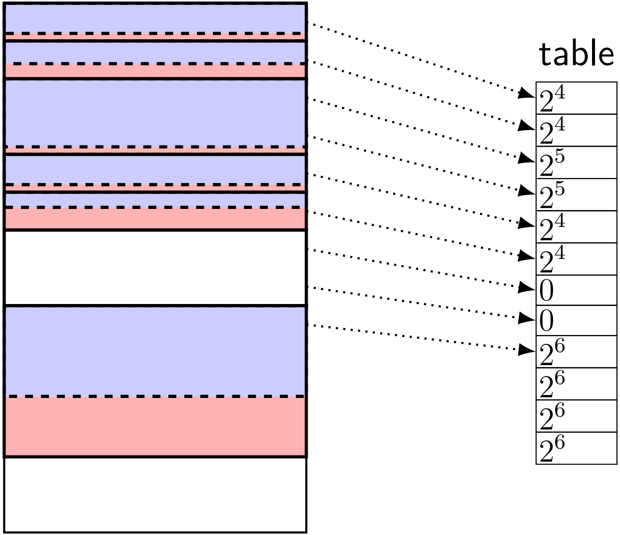


object allocated in power-of-two 'slots'

table stores sizes for each 16 bytes

addresses **multiples of size** (may require padding)

allocations and lookup table



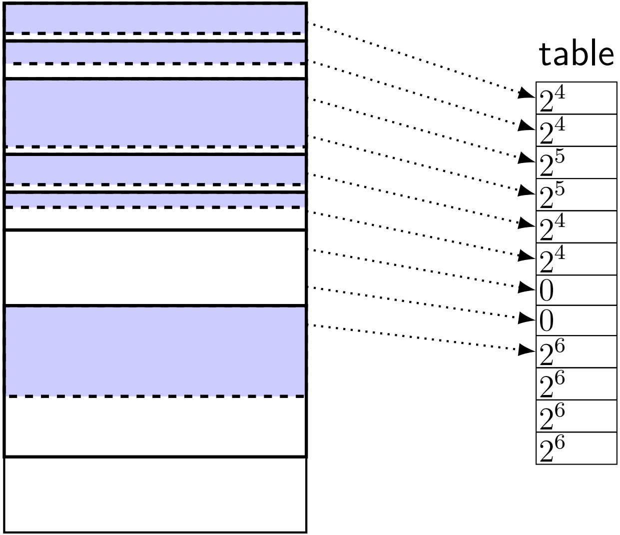
object allocated in
power-of-two ‘slots’

table stores sizes
for each 16 bytes

addresses multiples of size
(may require padding)

sizes are **powers of two**
(may require padding)

allocations and lookup table



object allocated in power-of-two ‘slots’

table stores sizes for each 16 bytes

addresses multiples of size (may require padding)

sizes are powers of two (may require padding)

managing the table

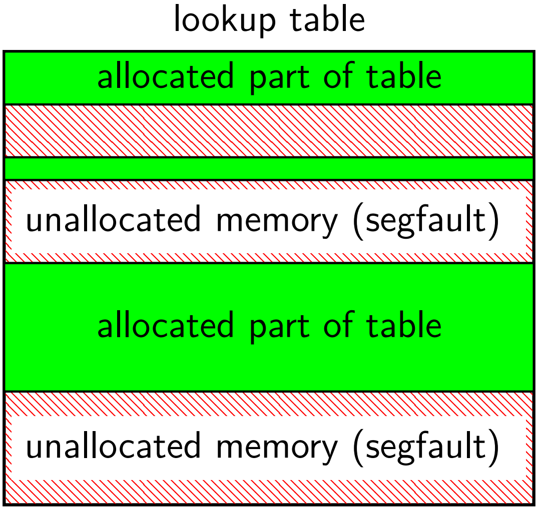
not just done `malloc()`/`new`

also for stack allocations:

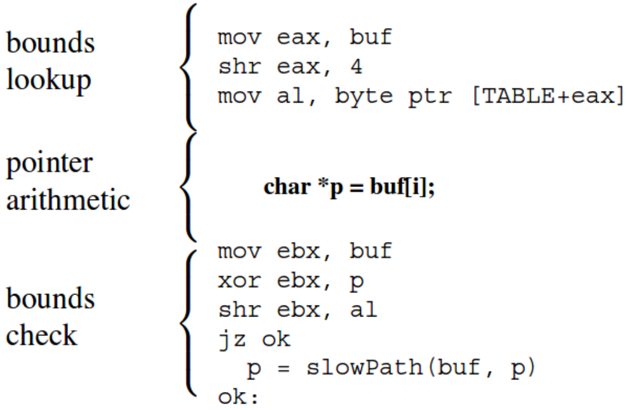
```
void vulnerable() {
    char buffer[100];
    gets(vulnerable);
}
```

```
vulnerable:
  // make %rsp a multiple
  // of 128 (2^7)
  andq $0xFFFFFFFFFFFFFF80, %rsp
  // allocate 128 bytes
  subq $0x80, %rsp
  // rax ← rsp / 16
  movq $rsp, %rax
  shrq $4, %rax
  movb $7, TABLE(%rax)
  movb $7, TABLE+1(%rax)
  ...
  movq %rsp, %rdi
  call gets
  ret
```

sparse lookup table



baggy bounds check: added code



```
mov eax, buf
shr eax, 4
mov al, byte ptr [TABLE+eax]

    char *p = buf[i];

mov ebx, buf
xor ebx, p
shr ebx, al
jz ok
  p = slowPath(buf, p)
ok:
```

Figure 5: Code sequence inserted to check unsafe pointer arithmetic.

baggy bounds check: added code

```
/* bounds lookup */
    mov buf, %rax
    shr %rax, 4
    mov LOOKUP_TABLE(%rax), %al
/* array element address computation */
    ...    // char * p = buf[i];
/* bound check */
    mov buf, %rbx
    xor p, %rbx
    shr %al, %rbx
    jz  ok
    ...    // handle possible violation
ok:
```

adapted from paper figure

avoiding checks

code not added if not array/pointer accesses to object

code not added when pointer accesses “obviously” safe

- author’s implementation: only checked within function

alternate approach: pointer tagging

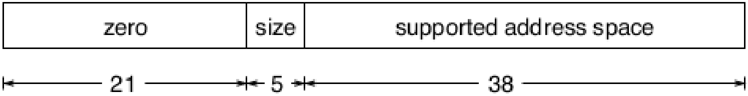
some bits of address are size

- replaces table entry/lookup

change code to allocate objects this way

works well on 64-bit — plenty of addresses to use

(c) Tagged pointer



baggy bounds performance

table: 4–72% time overhead (depends on benchmark suite)

table: 11–21% space overhead (depends on benchmark suite)

tagged pointers: slightly better on average

baggy bounds performance

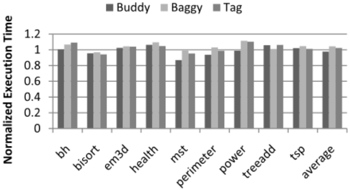


Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

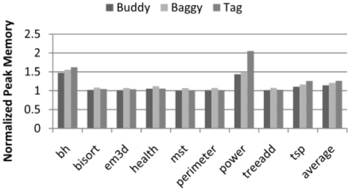


Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

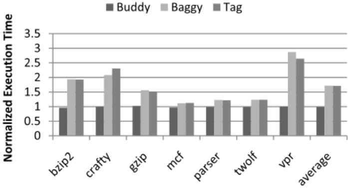


Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

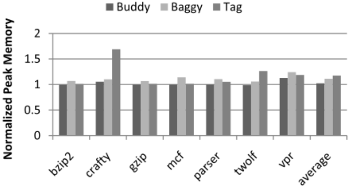


Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

benign out-of-bounds

baggy bounds also has support for benign bounds violations:

```
int rawArray[100];
int *array = &rawArray[-1];
// now pretend array's first index is 1
```

yes, this is done in real C programs

missing from baggy bounds

detecting use-after-free bugs

    or other cases of type confusion

detecting errors within an object:

```
struct Foo {
    char buffer[100];
    void (*danger)();
};
```

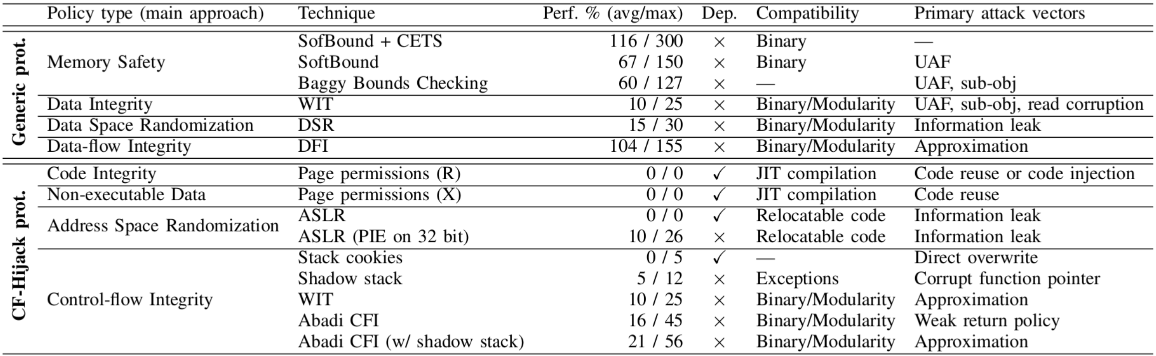
very fancy compiler analyses to eliminate checks

2013 memory safety landscape

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

2013 memory safety landscape



	Policy type (main approach)	Technique	Perf. % (avg/max)	Dep.	Compatibility	Primary attack vectors
Generic prot.	Memory Safety	SofBound + CETS	116 / 300	×	Binary	—
		SoftBound	67 / 150	×	Binary	UAF
		Baggy Bounds Checking	60 / 127	×	—	UAF, sub-obj
	Data Integrity	WIT	10 / 25	×	Binary/Modularity	UAF, sub-obj, read corruption
	Data Space Randomization	DSR	15 / 30	×	Binary/Modularity	Information leak
	Data-flow Integrity	DFI	104 / 155	×	Binary/Modularity	Approximation
CF-Hijack prot.	Code Integrity	Page permissions (R)	0 / 0	✓	JIT compilation	Code reuse or code injection
	Non-executable Data	Page permissions (X)	0 / 0	✓	JIT compilation	Code reuse
	Address Space Randomization	ASLR	0 / 0	✓	Relocatable code	Information leak
		ASLR (PIE on 32 bit)	10 / 26	×	Relocatable code	Information leak
	Control-flow Integrity	Stack cookies	0 / 5	✓	—	Direct overwrite
		Shadow stack	5 / 12	×	Exceptions	Corrupt function pointer
		WIT	10 / 25	×	Binary/Modularity	Approximation
		Abadi CFI	16 / 45	×	Binary/Modularity	Weak return policy
		Abadi CFI (w/ shadow stack)	21 / 56	×	Binary/Modularity	Approximation

Table II

THIS TABLE GROUPS THE DIFFERENT PROTECTION TECHNIQUES ACCORDING TO THEIR POLICY AND COMPARES THE PERFORMANCE IMPACT, DEPLOYMENT STATUS (DEP.), COMPATIBILITY ISSUES, AND MAIN ATTACK VECTORS THAT CIRCUMVENT THE PROTECTION.

alternative techniques

memory error detectors — to help with software testing
- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either
- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

alternative techniques

memory error detectors — to help with software testing

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

AddressSanitizer

like baggy bounds:

- big lookup table
- lookup table set by memory allocations
- compiler modification: change stack allocations

unlike baggy bounds:

- check reads/writes (instead of pointer computations)
  - only detect errors that read/write between objects
  - deliberate padding added to detect errors
- no power-of-two restriction
- table has info for every single byte (more precise)

Valgrind Memcheck

similar to AddressSanitizer — but no compiler modificaitons

instead: is a virtual machine

can't reliably detect stack errors

but works on unmodified binaries

alternative techniques

memory error detectors — to help with software testing
- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either
- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

automatic testing tools

basic idea: generate lots of random tests — “fuzzing”

look for segfaults and/or run with memory error detector

blackbox:

- just try random testing

whitebox:

- generate tests by looking at what program does internally

alternative techniques

memory error detectors — to help with software testing

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

static analysis

analyze program code directly

some overlap with whitebox testing

complete versus sound
- complete: no false positive
  - says memory error — actually a memory error
- sound: no false negative
  - says no memory error — actually no memory errors

many real analyzers neither complete nor sound

sometimes assisted by programmer annotations
- e.g. “this pointer should not be null”

alternative techniques

memory error detectors — to help with software testing
- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either
- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

better programming languages

get better information from programmer

ideal: eliminate memory errors without making program slower

some overlap with static analysis

- information used to prove no memory errors

example: “smart pointer” libraries for C++

example: Rust

other kinds of bugs?

many of these techniques work for other security bugs

testing, static analysis, programming language improvements

same basic ideas also applicable

plans for the future

assignment using a “fuzzing” tool

would like to go over some additional topics:

- command injection bugs
- web browser security
- whitebox fuzzing (‘informed’ random testing)
- better programming languages — Rust

I am flexible — different topics you want?

- sandboxing (another mitigation)
- synchornization-related security bugs
- static analysis?
- new mitigations proposed in research?
- other?