Heap Layout Optimisation For Exploitation

Sean Heelan

University of Oxford

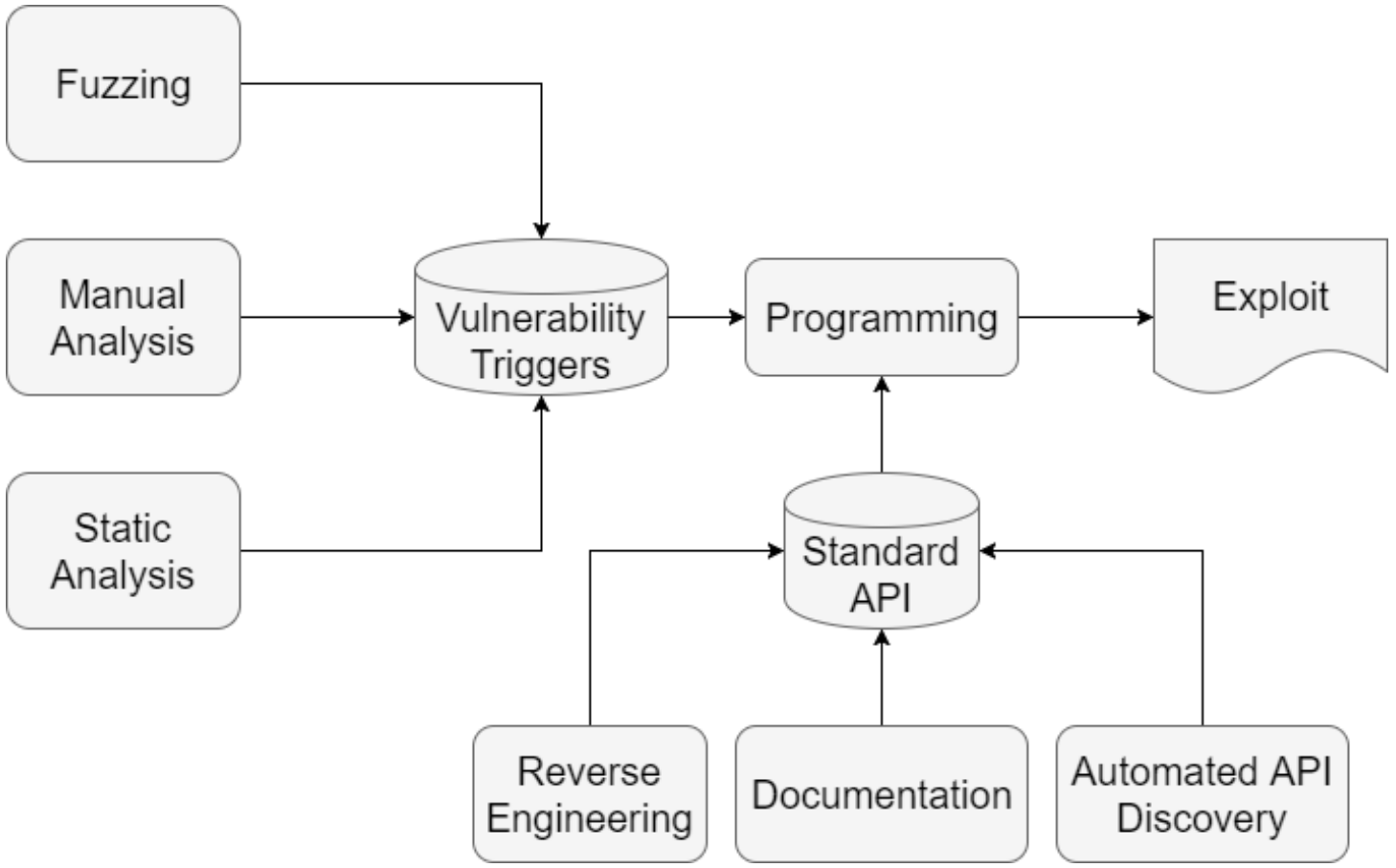
https://sean.heelan.io /@seanhn / sean@vertex.re

About Me

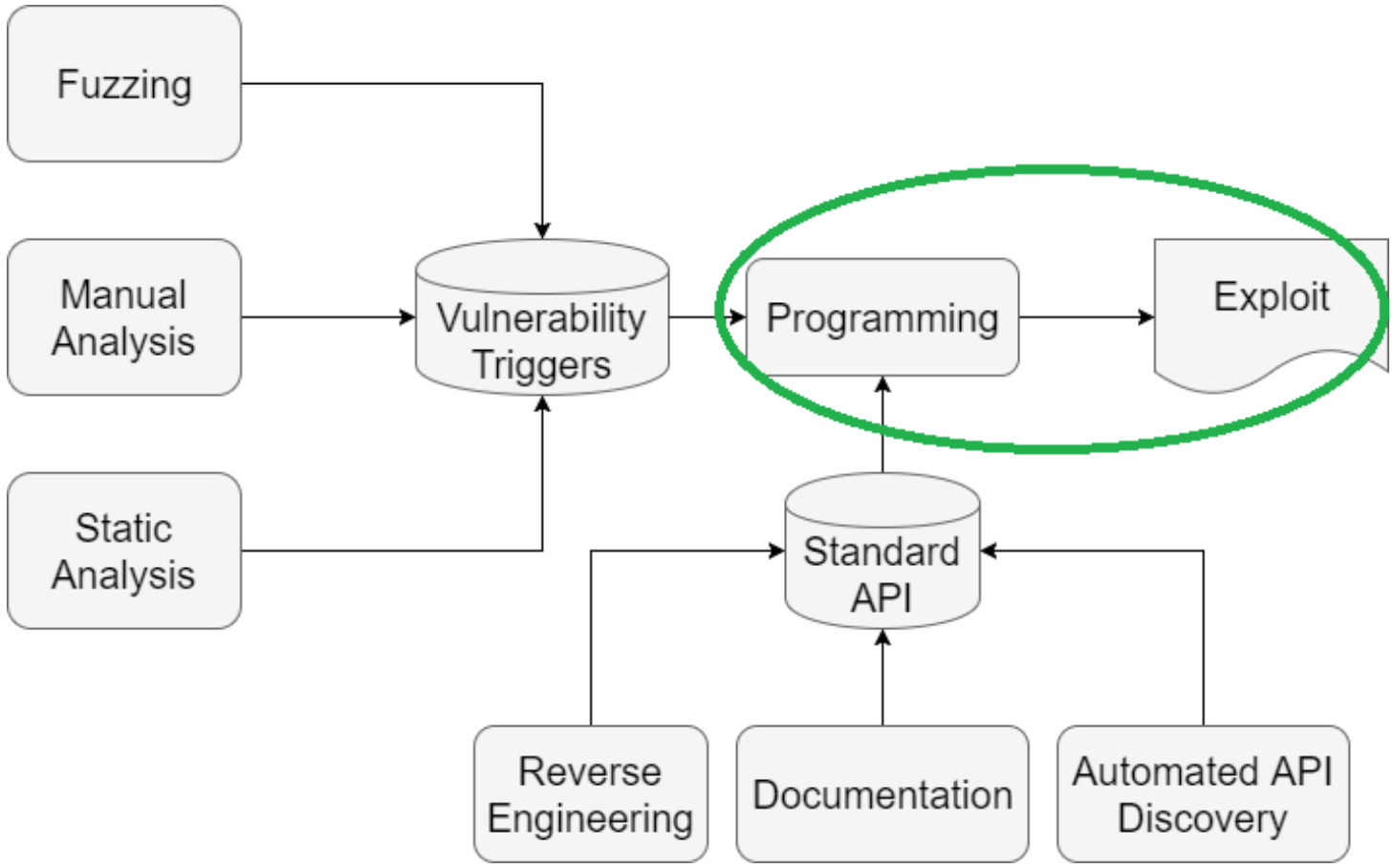
- Former Senior Security Researcher @ Immunity Inc
  - Vulnerability discovery and exploit development
  - Program analysis research, mostly focused on symbolic execution
- Former Founder + Director of Persistence Labs
  - Tool development for reverse engineering and binary analysis
- Currently a PhD candidate @ University of Oxford
  - Automation of exploitation for heap-related vulnerabilities

Introduction

Exploit Development Process



Exploit Development Process



Exploitation is Programming

- Utilising the program's 'standard' API as well as an API constructed from vulnerabilities in order to manipulate the program's state
- API constructed from vulnerabilities often referred to as 'primitives'
  - Read, write or execute
  - Utilise a vulnerability to provide the exploit developer with some controlled capability, e.g. write four arbitrary bytes at an arbitrary address
  - Used as the building blocks of an exploit, along with the standard API
- Primitives are usually constructed by manipulating the system into a particular state, then triggering a vulnerability

Constructing Primitives

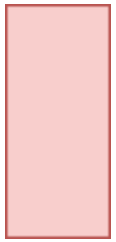
```
typedef struct {
    char *name;
    …
} User;

void rename(User *u, char *n)
{
    …
    strcpy(u->name, n);
}
void display(User *u) {
    print(u->name);
    …
}
```

- `User` objects are dynamically allocated and contain a pointer to a data buffer, which is also dynamically allocated (i.e. both are on the heap)
- `rename` has an overflow out of the `u->name` buffer
- `display` accesses the `name` field of the provided `User` object and prints whatever is there back to the user of the API
- How do we build a ‘read’ primitive from this?

Constructing Primitives

- What happens if we just trigger the vulnerability?
  - Entirely depends on the heap state.
  - Whatever is after the `User->name` buffer will be accessed/corrupted
  - Could be an unmapped paged, allocator metadata, any dynamically allocated application data

Constructing Primitives



Unmapped

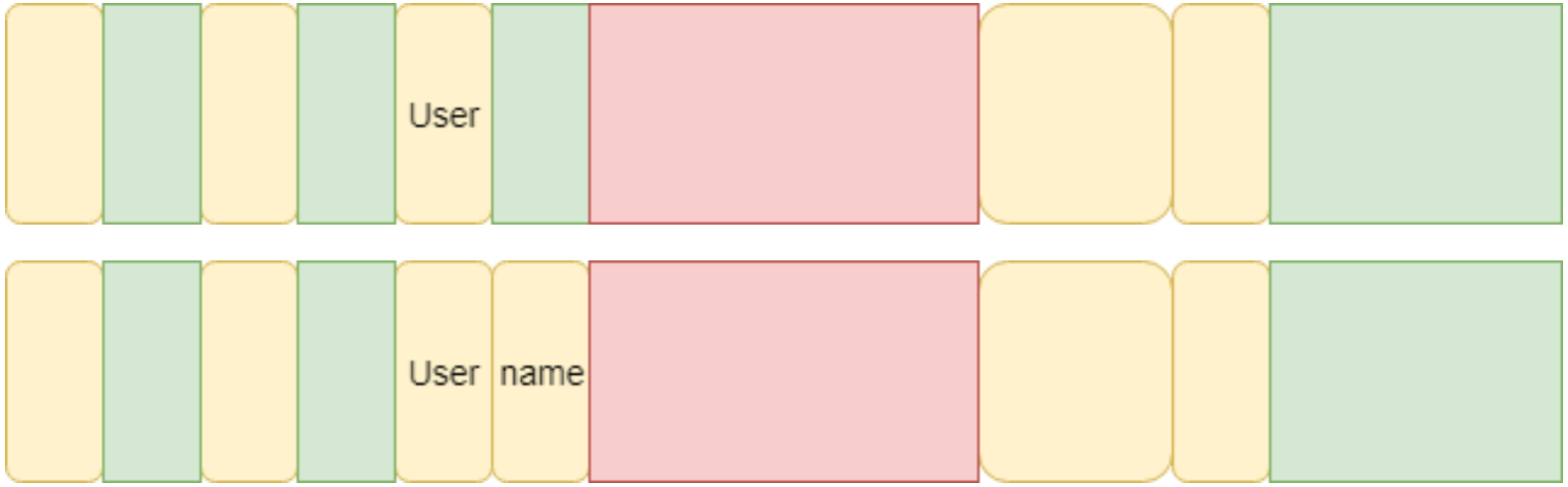
Free

In use

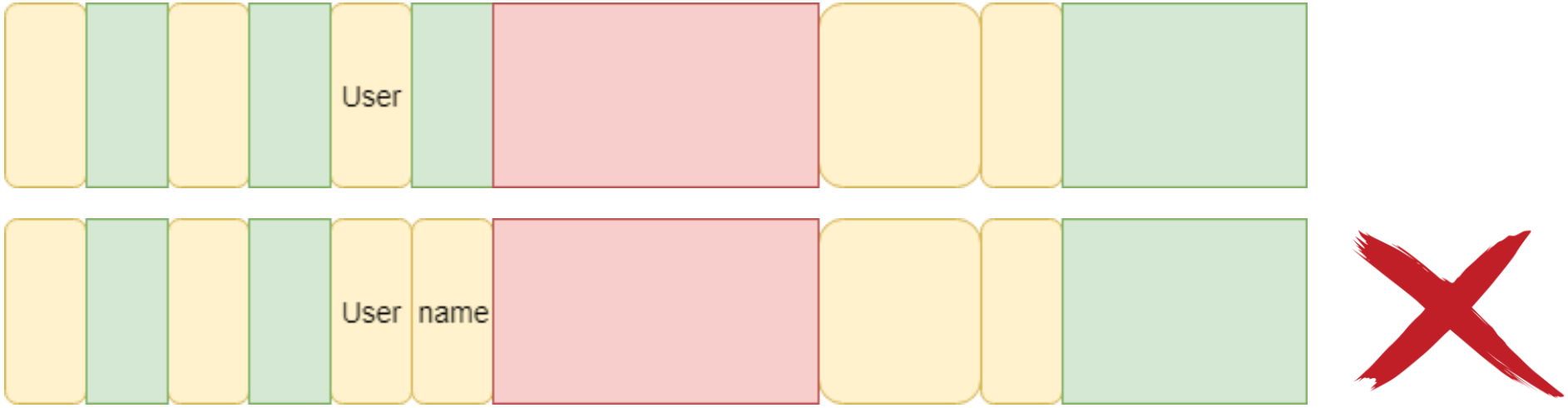
Goal: Position the name buffer immediately prior to the `User` object, such that when the vulnerability is triggered the `name` pointer in the `User` object is corrupted

Constructing Primitives

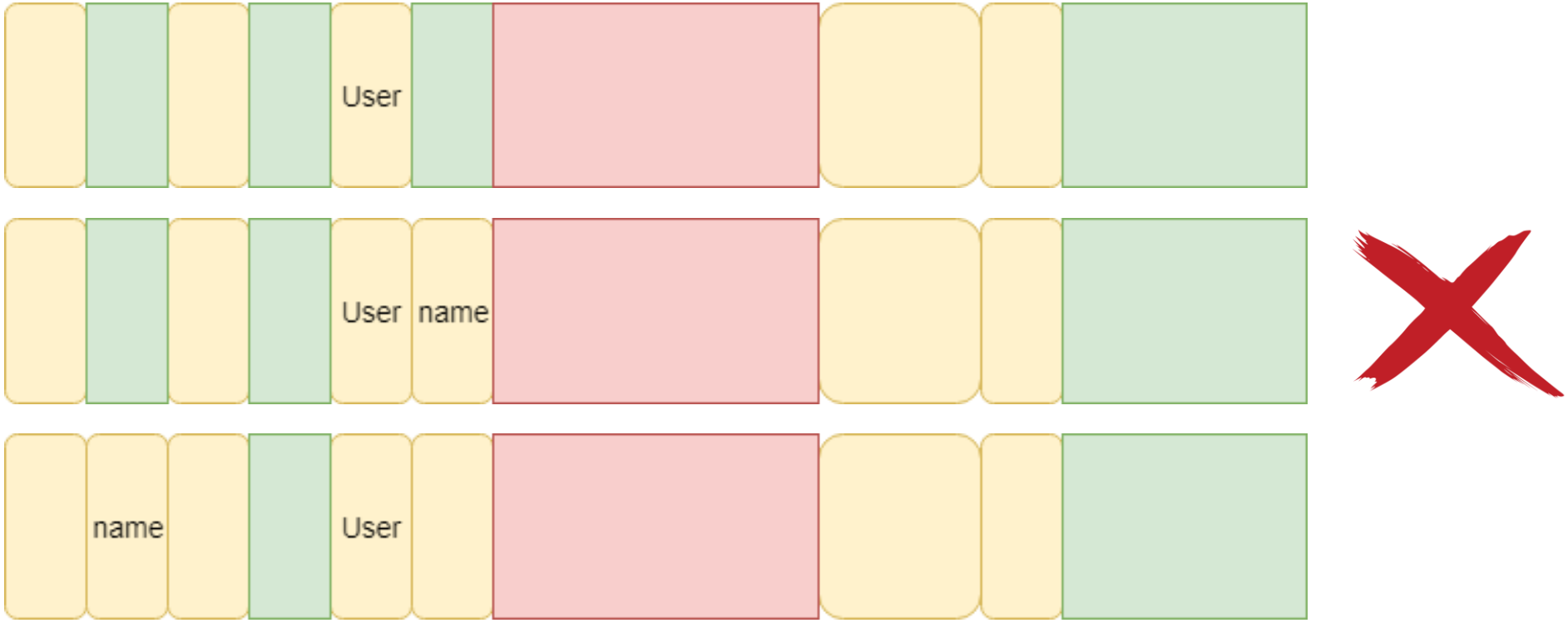
Constructing Primitives



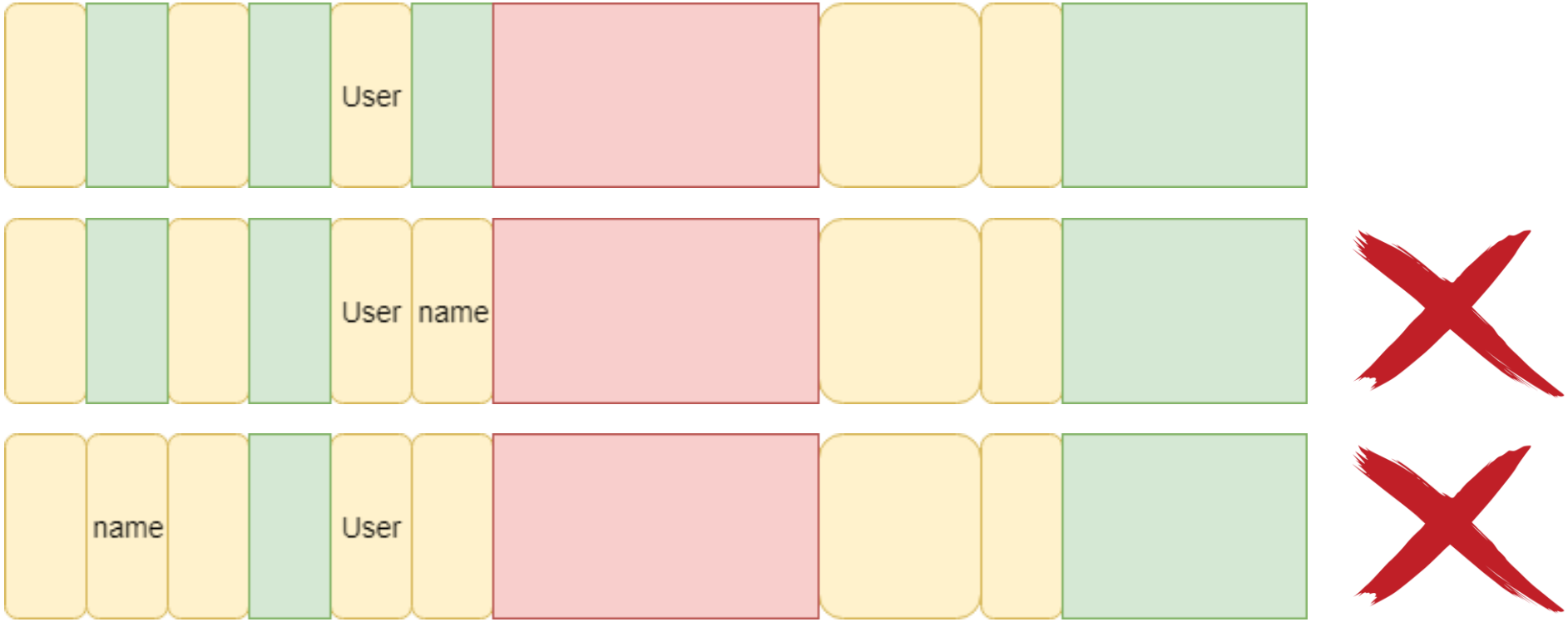
Constructing Primitives



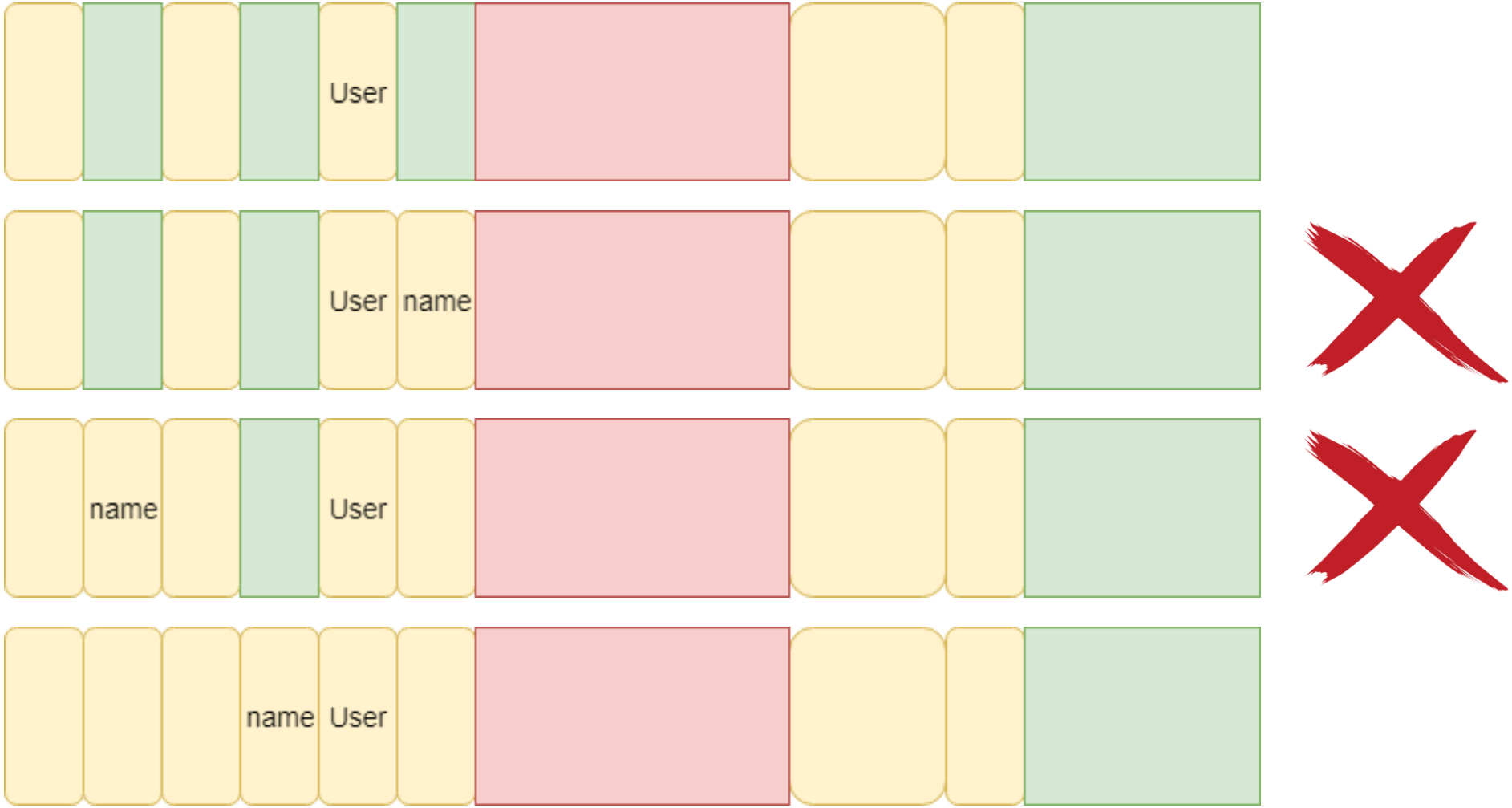
Constructing Primitives



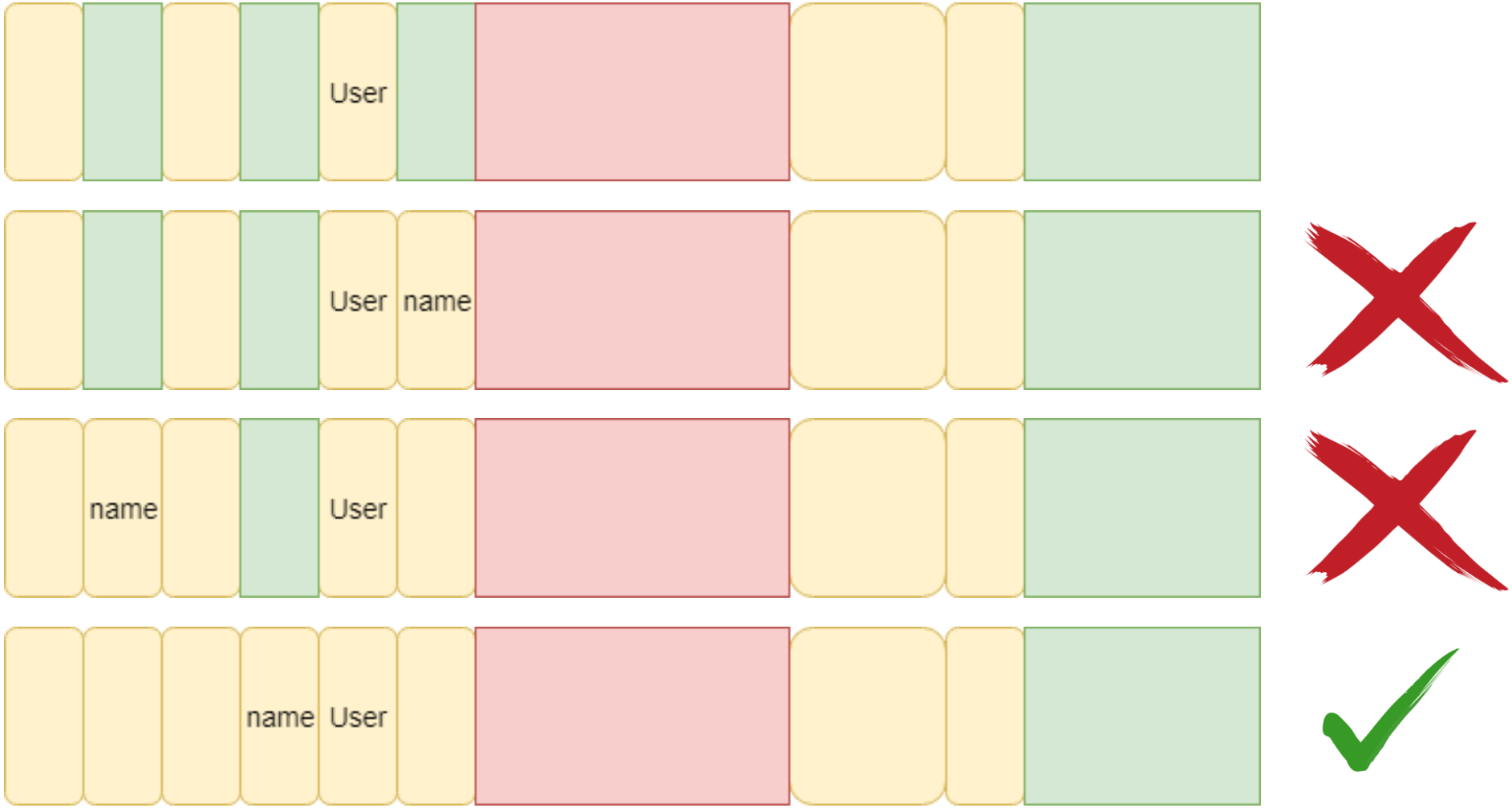
Constructing Primitives



Constructing Primitives



Constructing Primitives



Constructing Primitives

rename(…, ‘ABCDABCDABCDABCD\x41\x41\x41\x41’)

Results in User->name = 0x41414141

display(…)

Instead of printing the contents of the ‘name’ buffer, prints the data at 0x41414141

Constructing Primitives

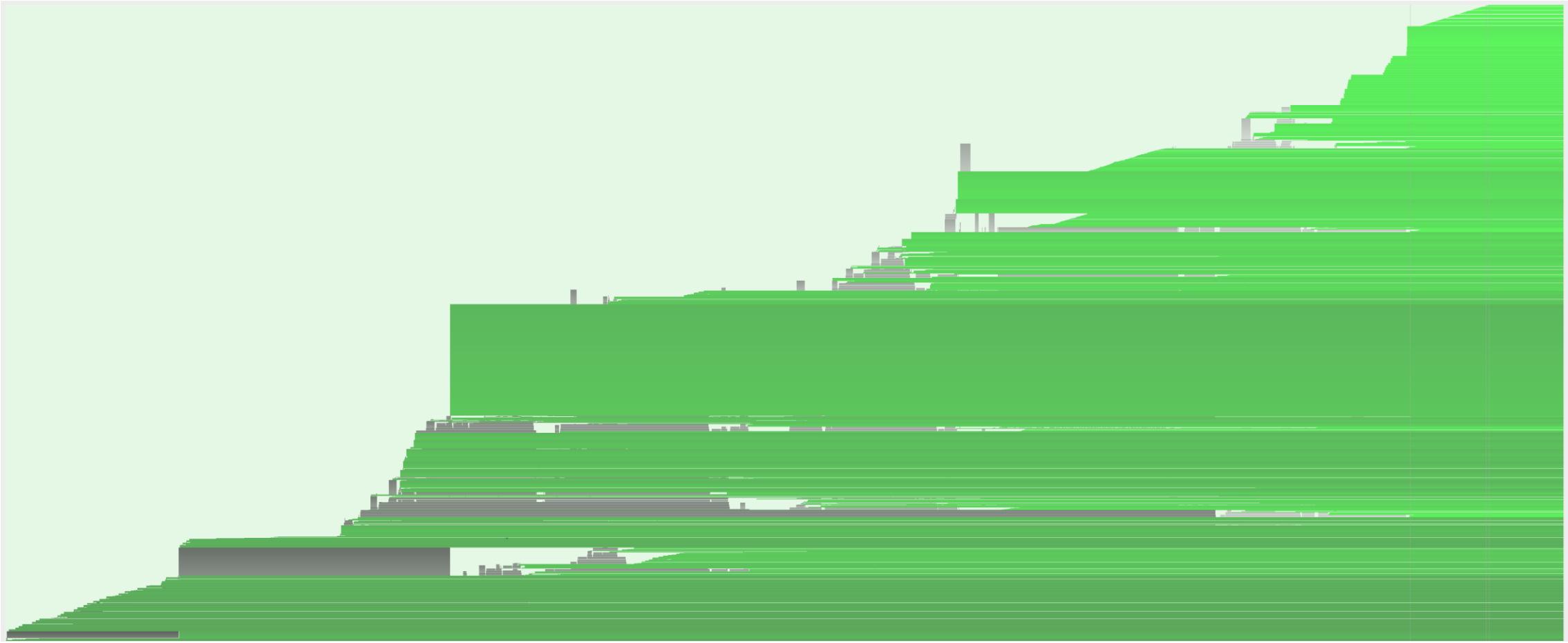
- Final primitive might look as on the right
- Function which the exploit developer can call whenever they need to read data at a particular address from the target process
- Reusable and predictable

```
read_data(addr) {
    manipulate_heap();
    allocate_user();
    trigger_vuln(addr);
    call_display();
}
```

Constructing Primitives

- For heap-based overflows constructing a viable primitive requires one to perform heap layout manipulation to ensure the right thing is corrupted
- Usually a labour intensive, manual task
  - Analyst needs to understand the allocator and the manner in which the application uses it
  - Then, given a starting state, needs to utilise the application's API to carefully craft a heap state
- Automating this is what we will focus on today

Reality …



Problem Overview

Physical Heap Layout Optimisation (HLO)

- Source buffer, `S`
  - The buffer from which the overflow or underflow originates once the vulnerability is triggered
- Destination buffer, `D`
  - The buffer which we wish to corrupt once the vulnerability is triggered
- Heap Layout Optimisation
  - Minimise the objective function `abs(S - D)`
  - The search for a sequence of inputs to a program such that when a vulnerability is triggered the minimal amount of collateral data is corrupted

Problem Setting & Restrictions

- Deterministic setting
  - The allocator's behaviour must be deterministic
  - The attacker must be able to determine the starting state of the heap, or (re)set it to a known state
  - There are no other actors interacting with the allocator, or the processes address space, at the same time (or if there is then their actions are deterministic)
- Physical layout, not logical
  - We are solving for relative ordering over addresses in memory, not allocation order

To Solve

- Automatically discover how to interact with the allocator via the application's API

To Solve

- Automatically discover how to interact with the allocator via the application's API
- Automatically discover how to allocate 'interesting' data on the heap via the application's API, i.e. figure out what to use as the destination buffer

To Solve

- Automatically discover how to interact with the allocator via the application's API
- Automatically discover how to allocate 'interesting' data on the heap via the application's API, i.e. figure out what to use as the destination buffer
- Automatically discover how to place the source buffer adjacent to the destination buffer such that when the vulnerability is triggered the data of interest in the destination buffer is corrupted

Challenges

- Allocators do not provide an API which allows one to specify relative positioning

Challenges

- Allocators do not provide an API which allows one to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms

Challenges

- Allocators do not provide an API which allows one to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use

Challenges

- Allocators do not provide an API which allows one to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use
- Interaction sequences which can be triggered via the application's API are often limited in various ways and 'noisy'

Challenges

- Allocators do not provide an API which allows one to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use
- Interaction sequences which can be triggered via the application's API are often limited in various ways and 'noisy'
- The search space across all interaction sequences is usually astronomically large

Heap Allocation Policies and Mechanisms

Heap Allocation Mechanisms & Policies

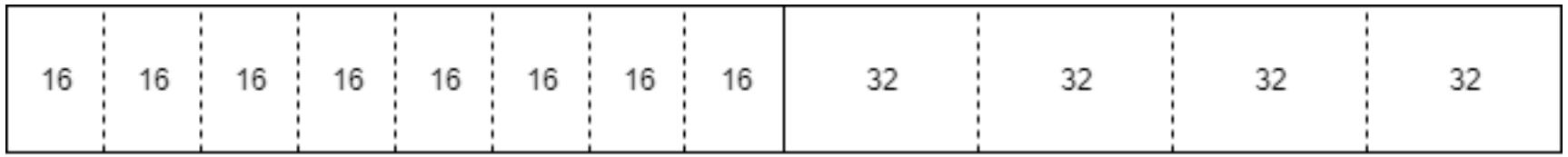
- "*The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified*" – ANSI C specification
- Location of a buffer used to service an allocation request is a product of the requested size, the current heap state and the **data structures and algorithms** used in the allocator

Heap Allocation Mechanisms and Policies

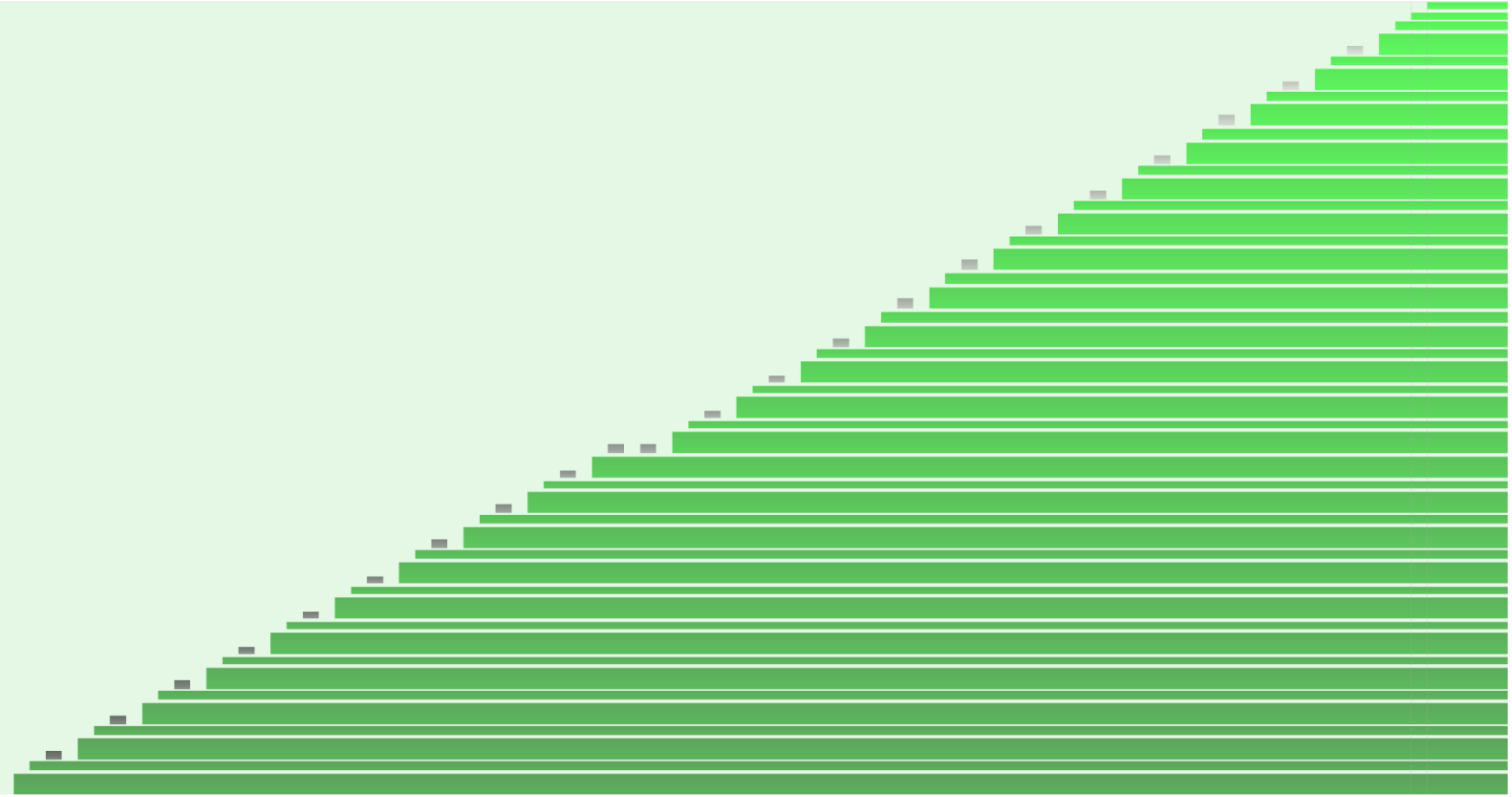
- Significant in determining the complexity of solutions
  - Splitting policy
  - Coalescing policy
  - Usage, or not, of segregated storage
  - Usage, or not, of non-determinism
  - Treatment of larger allocations
- Less significant
  - Fits algorithm
  - Usage, or not, of segregated free lists

Segregated Storage

- Large contiguous areas of memory cut into chunks of the same size
  - Never split
  - Never internally coalesced
  - Only externally coalesced as a whole
- Except for the first and last chunks, all other chunks of that size can only be adjacent to chunks of the same size



Non-Segregated Storage



Segregated Storage

An Algorithm for Heap Layout Manipulation

An Algorithmic Approach

- As mentioned, the following are usually performed manually
  1. Figure out how to allocate something interesting on the heap to corrupt
  2. Figure out how to interact with the heap via the allocator's API
  3. Given the starting state, utilise the program's API to manipulate the heap to place the source and destination buffers adjacent to one another
- Working in reverse
  - If we know how to interact with the heap, can we design a search algorithm which can minimise the objective function `abs(S - D)`

Design Considerations

- Ideally should be black box
  - Don’t want to have to customise for each allocator or allocator category
  - Interaction at the ANSI C memory management API level (malloc, free, realloc, calloc)
  - Outputs/observed behaviour in the form of return values from the above functions, and OS-generic things like mapped pages

Design Considerations

- Random search may be feasible
  - Despite the astronomical search space if we consider all permutations of API calls and arguments the solution space has a lot of symmetry
  - We only care about relative positioning for the source and destination, not absolute positioning
  - We don’t care about the positioning of holes at all, only their existence

Black Box Random Search

```
function SEARCH(g, m, r)
    while g ≠ 0 do
        tmpInst = ConstructInstance(m, r)
        tmpDist = Execute(tmpInst)
        if tmpDist < minDist then
            dist ← tmpDist
            inst ← tmpInst
        end if
        g ← g − 1
    end while
    return inst
end function
```

Black Box Random Search

```
function SEARCH(g, m, r)
    while g ≠ 0 do
        tmpInst = ConstructInstance(m, r)
        tmpDist = Execute(tmpInst)
        if tmpDist < minDist then
            dist ← tmpDist
            inst ← tmpInst
        end if
        g ← g − 1
    end while
    return inst
end function
```

```
function CONSTRUCTINSTANCE(m, r)
    result = NewInstance()
    i ← random(0, m − 1)
    while i ≠ 0 do
        if random(1, 100) < r then
            AppendAllocSequence(result)
        else
            AppendFreeSequence(result)
        end if
        i ← i − 1
    end while
    if random(0, 1) = 0 then
        AppendSrcSequence(result)
        AppendDstSequence(result)
    else
        AppendDstSequence(result)
        AppendSrcSequence(result)
    end if
    return result
end function
```

Automatic HLO on Synthetic Problems

Experimental Setup

- *“Given a source buffer with size X, and a destination buffer with size Y, can we generate a sequence of inputs which places the source and destination immediately adjacent”*
- Source and destination sizes from the cross product of 0, 64, 512, 4096, 16384, 65536
- Each source/destination combination run from 4 different starting heap configurations
  - Captured the initial heap states of Python, Ruby, and both PHP’s own heap and the system heap after PHP’s startup

Noise

- Often no way to trigger a single allocator interaction via the program's API
  - E.g. when a string is created the target might malloc a size we control but also allocate one or more other related objects
- These noisy interactions can impact the heap layout negatively and make it harder to achieve our desired layout
- To mimic this challenge we run three versions of each experiment with a varying number of noisy allocations taking place between the allocation of the source and the allocation of the destination
  - No noise, one noisy allocation and four noisy allocations

Experiments

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00

- 'Und' – Underflow, 'Ovf' – Overflow
- % < 4096 – Percentage of experiments ending with source and destination within a page of each other
- % Optimal – Percentage of experiments ending with source and destination direct adjacent
- Error – Normalized error. Ratio of # of bytes between source and destination to the larger of the source or destination
- Max 500,000 instances per experiment (avg. < 5 mins of run time with 40 concurrent workers)

Experiments

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00

- Each experiment consists of challenging the search to find a series of allocator interactions which place the source and destination adjacent
- There are 36 different source and destination size permutations and 4 different heap starting states
- No segregated storage and no noise, almost 100% success in all cases (source and destination directly adjacent).
- A single sub-optimal result (still ends with source within 32 bytes of destination however)

Experiments

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00
tcmalloc-2.6.1	0	94	94	83	81	1.01	1.24

- Segregated storage clearly more difficult to deal with
- Percentage of optimal results drops, but is still above 80%
- Average error in sub-optimal cases is low – just over 1x, e.g. off by a single allocation with size equal to the source/destintation

Experiments

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00
tcmalloc-2.6.1	0	94	94	83	81	1.01	1.24
avrlibc-r2537	1	83	83	44	50	0.43	0.58
dlmalloc-2.8.6	1	81	83	22	58	0.37	0.68
tcmalloc-2.6.1	1	83	89	44	58	1.07	1.52

- Adding a single noisy allocation reduces the number of optimal results significantly, for both segregated storage and otherwise
  - Worst - dlmalloc, underflow – 97% to 22%
  - Best – tcmalloc, overflow – 81% to 58%
  - Overall, 44% optimal without segregated storage, 51% optimal with it
- Despite the drop in optimal results, the average error remains low
  - .52 without segregated storage, 1.3 with it

Experiments

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00
tcmalloc-2.6.1	0	94	94	83	81	1.01	1.24
avrlibc-r2537	1	83	83	44	50	0.43	0.58
dlmalloc-2.8.6	1	81	83	22	58	0.37	0.68
tcmalloc-2.6.1	1	83	89	44	58	1.07	1.52
avrlibc-r2537	4	75	78	33	42	1.55	1.65
dlmalloc-2.8.6	4	75	78	14	44	1.25	1.95
tcmalloc-2.6.1	4	72	83	22	53	1.40	2.47

- Further increasing the noise results in continued decrease in the number of optimal results and an increase in the average error
- Without segregated storage 33% optimal, avg. error 1.6
- With segregated storage 38% optimal, avg. error 1.94

Experimental Summary

- No noise and without segregated storage, near perfect results (one suboptimal result but with a very low error)
- No noise and segregated storage, optimal results in more than 80% of cases, average error around 1x
- Increasing the amount of noise does have a significant impact
  - With 4 noisy allocations then the number of optimal results drops into the 30-40% region
  - However, this is with < 5mins of analysis time per experiment on average
- Overall, black box random search seems promising for a significant number of problem instances!

Automatic HLO for Real Programs

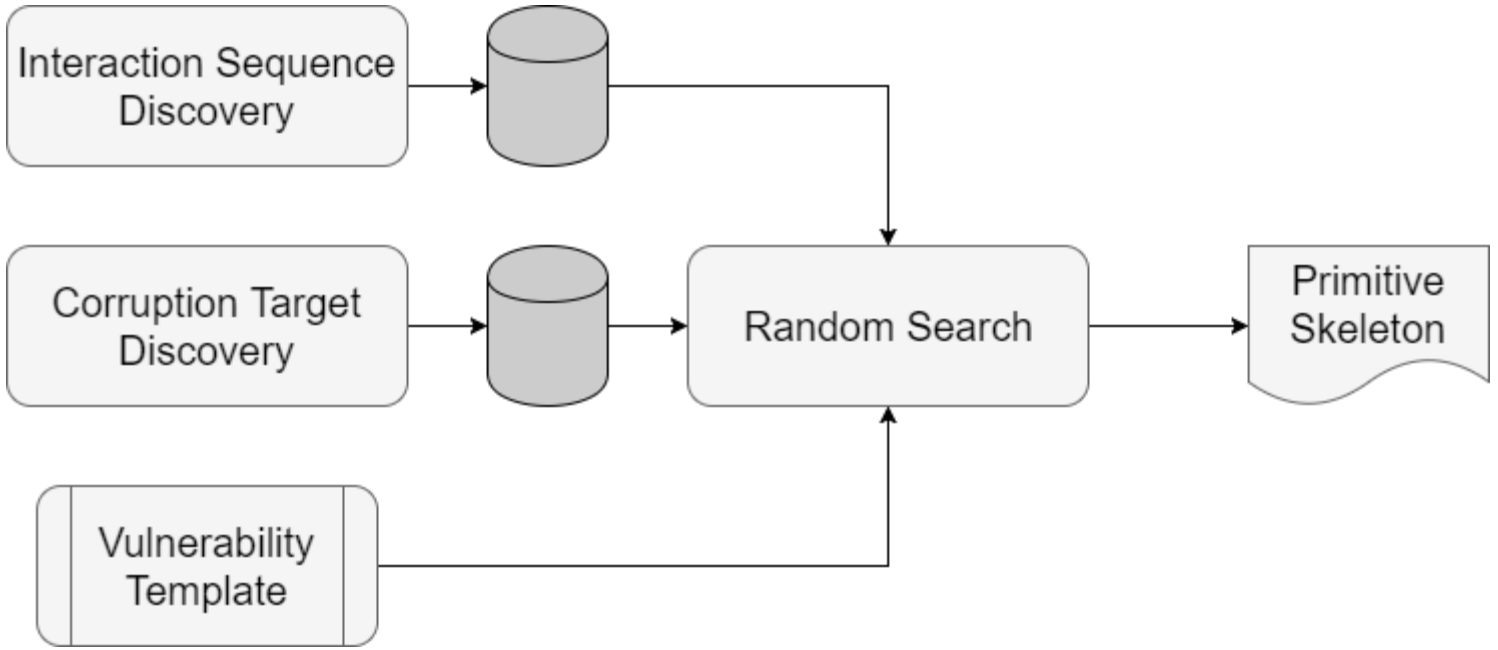
Automatic HLO for Real Programs

- So the random search at least looks promising. What do we need to do it for a real target?
  - Knowledge of how to trigger different allocator interaction sequences via the target's API
  - (Optionally) A means to discover how to allocate 'interesting' target data on the heap
  - An implementation of the search

Automatic HLO for Real Programs

- To evaluate this idea we choose the PHP language interpreter
- Large, modern application written in C with a non-trivial interface for manipulating its internal state (programs written in the PHP language)
- Our tool will therefore be writing PHP programs for us which perform heap layout manipulation

Architecture



Why a ‘Skeleton’?

- Output of the system is a PHP program which guarantees the relative positioning of the source and destination and triggers the vulnerability
- Does not guarantee that the vulnerability will corrupt the destination however
  - E.g. if the system says the destination is 8 bytes after the source it is up to the user to ensure that the vulnerability is triggered in such a way that 8 + X bytes of data after the source are corrupted
- Does not discover how to use the corrupted data
  - The system guarantees positioning, the vulnerability trigger provides the corruption, it is up to the user to figure out how to trigger the use of the corrupted data in order to complete the primitive

Identifying Available Interaction Sequences

- We want to synthesise fragments of PHP code which trigger unique interaction sequences with its allocator
- Correlating PHP code with the interaction sequences which are triggered is straightforward
- Automatically synthesising valid PHP which triggers useful interaction sequences is a bit more involved

Synthesising PHP Fragments

- Effectively performed by fuzzing, tuned towards discovering interaction sequences rather than bugs
  - Leverages prior work "*Ghosts of Christmas Past: Fuzzing Language Interpreters using Regression Tests*" from Infiltrate '14
- Basic idea is to deconstruct PHP's regression tests into small, valid, chunks of PHP code. Then utilise mutation and recombination to produce new fragments with new behaviours.

Fuzzing for Allocator Interactions

- Mutations are intended to produce new interaction sequences not crashes
  - E.g. fuzzers will typically replace integers with values 0, $2^{32} - 1$, $2^{31} - 1$ intended to hit edge cases. Instead we use integer values and string lengths that have a relation to allocation sizes we have not previously seen.
- Our measure of fitness is not based on code coverage but instead based on whether or not a new allocator interaction sequence is produced
- We discard any fragments that result in the interpreter exiting with an error
- We favour the shortest, least complex fragments, with priority being given to fragments with a single function call

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, ‘Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
      $image, $gaussian, 16, 0));
?>
```

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, ‘Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
      $image, $gaussian, 16, 0));
?>
```

```
imagecreatetruecolor(180, 30)
imagestring($image, 5, 10, 8, ‘Text’, 0x00ff00)
array(array(1.0, 2.0, 1.0), array(2.0, 4.0, 2.0))
array(1.0, 2.0, 1.0)
array(2.0, 4.0, 2.0)
var_dump(imageconvolution($image, $gaussian, 16, 0))
```

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, ‘Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
      $image, $gaussian, 16, 0));
?>
```

```
imagecreatetruecolor(180, 30)
imagestring($image, 5, 10, 8, ‘Text’, 0x00ff00)
array(array(1.0, 2.0, 1.0), array(2.0, 4.0, 2.0))
array(1.0, 2.0, 1.0)
array(2.0, 4.0, 2.0)
var_dump(imageconvolution($image, $gaussian, 16, 0))
```

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

Synthesis

imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)

Synthesis

imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)

imagecreatetruecolor(1, 1)
imagecreatetruecolor(1, 2)
imagecreatetruecolor(1, 3)
imagecreatetruecolor(1, 4)
…

Synthesis

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

```
imagecreatetruecolor(1, 1)
imagecreatetruecolor(1, 2)
imagecreatetruecolor(1, 3)
imagecreatetruecolor(1, 4)
…
```

```
imageconvolution(array(1.0, 2.0, 1.0), imagecreatetruecolor(180, 30), 16, 0)
imageconvolution(array(2.0, 4.0, 2.0), imagecreatetruecolor(180, 30), 16, 0)
imageconvolution(imagecreatetruecolor(180, 30), imagecreatetruecolor(180, 30), 16, 0)
…
```

Synthesising PHP Fragments

- From PHP's 12k or so tests we produce 300 standalone fragments containing a single function call
- 15 minutes or so of fuzzing (80 cores) produces over 10k fragments which trigger unique allocator interaction sequences
- Varying in length from a single allocator interaction to thousands of allocator interactions per fragment

Black Box Random Search for PHP

- The CONSTRUCT function now pieces together fragments of PHP
- We augment PHP with functionality which allows us to query the distance between the source and destination pointers, so we can get a value for the objective function
- SEARCH is a search over the space of PHP programs for one which minimises the distance between the source and destination

Finding Interesting Corruption Targets

- Instead of expecting the user to tell us how to allocate the corruption target we can try to automate the process
- What counts as ‘interesting’?
  - Program specific things might include permission bits, or flags indicating what features of the target are enabled in the current context
  - We will focus on something more general however – pointers!
- Goal of the system is then to produce an input such that a specific pointer is corrupted by the vulnerability
  - When used, offers the exploit developer a read/write/execute primitive

Finding Interesting Corruption Targets

- As before, we take as input the fragments extracted from the PHP tests
- Execute each in isolation and record the addresses of all buffers allocated which survive beyond the end of the fragment
- For each allocated buffer scan over its contents and see if anything looks like it might be a pointer based on heuristics
  - Alignment
  - Value
- Final filter based on mapped addresses

Vulnerability Templates

- We now have a means to search over heap configurations as well as identify interest corruption targets
  - Finally, we need a way for the user to provide a vulnerability trigger
- Each vulnerability is likely to require different setup in order to trigger, which we don’t want to hardcode
- Instead, we allow users to specify a vulnerability template which will be completed with the code required to perform the heap manipulation and allocate the target

Vulnerability Templates

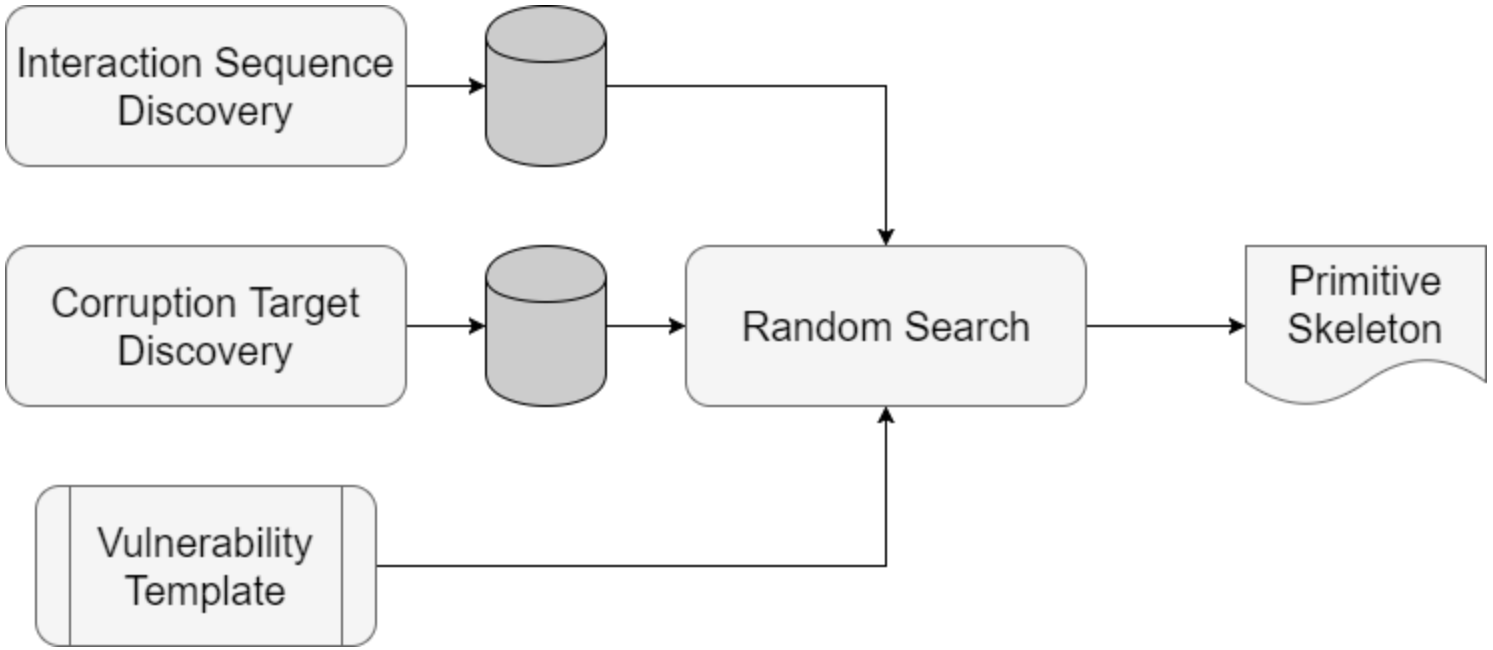
```
<?php
$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
?>
```

Vulnerability Templates

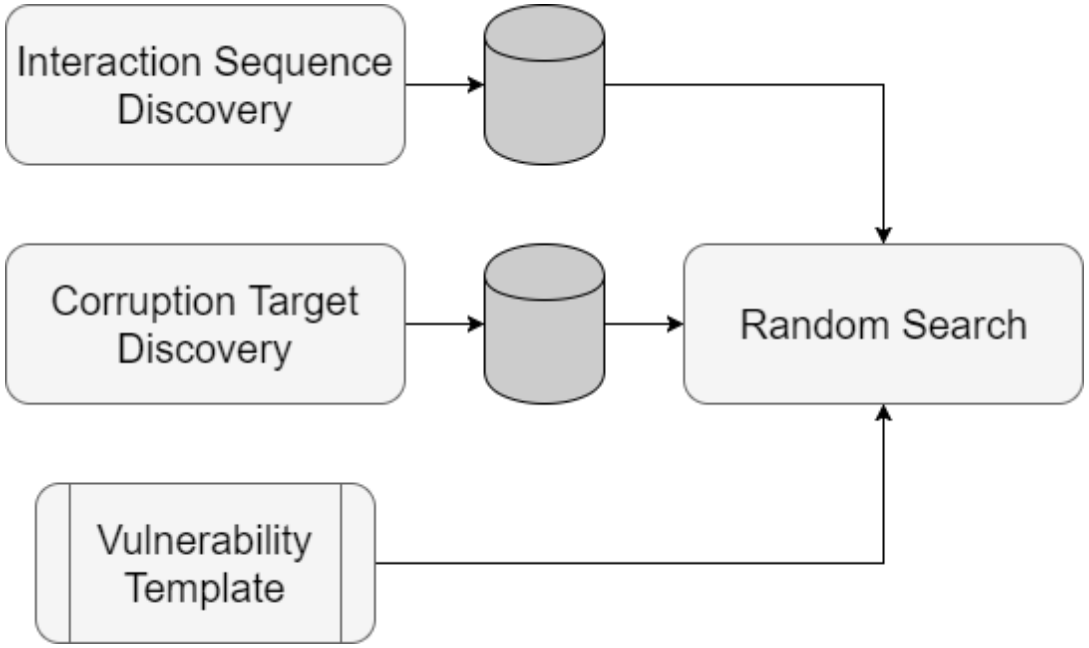
```php
<?php
$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
?>
```

```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# BEGIN-DST-CREATE
shrike_record_destination(1);
$dst = imagecreatetruecolor(1, 1);
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

Architecture



Architecture




```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# <...>
$var_vtx_24 = imagecreatetruecolor(1, 1);
# <...>
$var_vtx_24 = 0;
$var_vtx_102 = imagecreatetruecolor(1, 1);
$var_vtx_103 = imagecreatetruecolor(2, 256);
$var_vtx_104 = str_repeat('A', 225);
$var_vtx_105 = str_repeat('A', 225);

# BEGIN-DST-CREATE
shrike_record_destination(1);
$vtx_dst = array_fill(0, 1, '..');
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

Vulnerability -> Targeted Corruption

```php
<?php
$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
?>
```

```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# BEGIN-DST-CREATE
shrike_record_destination(1);
$dst = imagecreatetruecolor(1, 1);
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# <...>
$var_vtx_24 = imagecreatetruecolor(1, 1);
# <...>
$var_vtx_24 = 0;
$var_vtx_102 = imagecreatetruecolor(1, 1);
$var_vtx_103 = imagecreatetruecolor(2, 256);
$var_vtx_104 = str_repeat('A', 225);
$var_vtx_105 = str_repeat('A', 225);

# BEGIN-DST-CREATE
shrike_record_destination(1);
$vtx_dst = array_fill(0, 1, '..');
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false,
    PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

Evaluation

- User provides as input a vulnerability template
- User receives as output a PHP program which places a destination object on the heap, containing a pointer to corrupt, manipulates the heap to get the corruption source as close as possible to it, and then triggers the vulnerability
- Parallel analysis on 40 cores
  - Only shared info is the minimum value achieved for the objective function
  - Approx. 12k instances evaluated per second
  - Max of 12 hours analysis time (approx. max of ½ billion instances evaluated)

Evaluation

CVE ID	Vulnerability Type	Component	Src. Size	Dst. Size	Noise	Initial Dist.	Final Dist.	Time to Best	Instances to Best
2015-8865	OOB Write	File Parsing	480	264	42	119360	0	3780s	3753540
2016-5093	OOB Read	String Manip.	544	264	1	39680	0	9s	8937
2016-7126	OOB Write	Image Manip.	1	264	48	384088	16	1838s	2157812

- Vulnerabilities are the first three finite, linear OOB access bugs that were encountered in the PHP bug tracker
  - Obviously not exhaustive proof of feasibility, but no ‘hard’ cases with worse results purposefully excluded =)
- Noise – The number of allocations other than the allocation of the source buffer which are triggered by the call to the function which allocates the source
- Initial Dist. – Distance from source to destination when the original vulnerability trigger is run
- Final Dist. – Distance from source to destination after the heap layout is crafted

Evaluation

CVE ID	Vulnerability Type	Component	Src. Size	Dst. Size	Noise	Initial Dist.	Final Dist.	Time to Best	Instances to Best
2015-8865	OOB Write	File Parsing	480	264	42	119360	0	3780s	3753540
2016-5093	OOB Read	String Manip.	544	264	1	39680	0	9s	8937
2016-7126	OOB Write	Image Manip.	1	264	48	384088	16	1838s	2157812

- 2/3 cases end with an optimal layout (source and destination immediately adjacent, no collateral damage when the vulnerability is triggered), despite PHP 7 using segregated storage and one of those cases having significant noise
- Case with error of 16 is due to two noisy 8 byte allocations being placed between the source and destination which the search fails to discover how to place elsewhere
- Noise results in an increase in the amount of time taken to find the best result

Conclusion

Takeaways

- Black box random search is an effective mechanism for automatic heap layout manipulation
- Segregated storage was the most significant allocator policy in terms of the difficulty of problem instances
- As the noise in interaction sequences increases, so does the difficulty of the problem

Future Work

- Improving the search algorithm
  - Quite a few synthetic instances which are unsolved via random search
  - Many are solved when provided more time, but many are not
- Lifting the determinism restriction
  - In particular, the ability to (re)set the heap to a known state, or discover its current state, is significant
  - Most likely will necessitate associating a probability of success with solutions
- Porting the search to other real-world targets
  - Is it generally easy to repurpose fuzzers as done for PHP?
  - Media players, JS engines, document viewers

Thanks / Questions?

Paper & Code: https://sean.heelan.io/research

@seanhn / sean@vertex.re