From Garbage to Gold[*]

Timing-sensitive garbage collection in practice

Mikael Blomstrand
Kungl. Tekniska Högskolan
mbloms@kth.se
Göteborgs Universitet
gusblomsmi@student.gu.se

Christoffer Olsson
Chalmers University of Technology
chol@student.chalmers.se

1 Introduction

Today there are several different practices that help keeping information secret. Firewalls, access control lists and cryptography can be great tools, but they can't stop mistakes made by programmers. Such mistakes can leave holes that could be exploited to leak information.

None of these practices make any assertions regarding how that information is later handled, or if it is propagated in some way. An example can be given with cryptography, in that it can securely deliver information without interference and in a safe manner, but once the information is delivered to the receiver, encryption can't save you from mistakes in how the receiver handles the information. A popular approach to solving this problem is information flow control [7].

Information flow control aims to make it more safe to execute untrusted code by analyzing information flows before (static analysis) and/or during (dynamic analysis) the execution. Data and variables are divided into different levels of security and the goal is to prove that high security level data can't leak into lower levels of security. This way confidential information, such as credit card numbers etc, can be guaranteed to never leave the trusted parts of a program.

Unfortunately, information flow control is not a silver bullet. The information flow model must define all ways information can flow from one place to another, and if a vulnerability is obscure enough, it could be exploited as a runtime side-channel attack without information flow control being able to catch it. Examples of such side-channels are CPU caches [5], schedulers [9], and lazy evaluation [1].

This project focuses on one such side-channel attack exploiting the garbage collector in a programming language with automatic memory management. The attack was discovered by Pedersen and Askarov [6] and exploits the fact that one can trigger the garbage collector by allocating memory and measure the time it takes to run it. The time can be used to infer what happened before the garbage collection run, and by extension conclusions about some sensitive value can be drawn.

Pedersen and Askarov [6] managed to construct examples of such an attack where they where able to reliably leak sensitive information at a rate of up to 1 B/s.

2 Background

2.1 Garbage Collection

The job of a Garbage Collector is simple. Free up memory so it can be reused again for any future process that might need it. There are many garbage collection algorithms, but perhaps the two most well known are Copying Garbage Collector (often shortened Copy Collector), and Mark-and-Sweep. In both algorithms, the idea is that any object that isn't transitively reachable can be reclaimed by the Garbage Collector.

2.1.1 Mark-and-Sweep

A Mark-and-Sweep garbage collector traverses the heap in two phases, the Mark phase and the Sweep phase.

The Mark phase traverses all reachable objects starting from a set of root pointers and marks them as in-use.

The Sweep phase then traverses all objects on the heap. All objects that are not marked as in-use

[*]Based on the paper "From trash to treasure" [6].

Language-based security, 2019, Group 35
June 3, 2019.

are freed from memory and the objects that are marked have their in-use flag unset.

2.1.2 Copying Garbage Collector

A copying garbage collector (often called "copy collector") traverses all reachable objects and copies them to a new location. After collection is done, the whole old memory region can be discarded. This is often implemented by dividing memory into two segments. During collection, segment that holds all the data is often called the *from-space* and new location is called the *to-space*. The tricky part in a copy collector is that pointers must be rewritten so that they point to the newly allocated data rather than the old data.

A disadvantage of the algorithm is that it uses more memory than a Mark-and-Sweep. This is because, during collection, it must hold both the old objects and the new copies in memory. The simplest implementation requires twice as much memory as a Mark-and-Sweep.

The advantage of a copy collector is that it compacts the allocated memory when copying it, making it faster and easier to allocate new memory. There's also no need to traverse the entire heap, and the less data still alive on the heap, the faster the collection.

In practice this can mean a lot, because in most programs, there are usually many allocations of small variables that are only used for a short time.

2.2 Timing Attack

A timing attack can shortly be described as a side-channel attack where an attacker attempts to obtain information by measuring the time that it takes to execute a task, such as a branch or a function call. If execution time depends on some variable, it is possible to figure out what this variable is, or at least bits of what it is.

An example is when the two branches of an if statement takes different amounts time. It's then possible to know whether the condition evaluated to true or false based on the execution time.

If the condition is something like `if (x > secret)` and we control `x`, then `secret` can be found by binary search, leaking at least one bit at a time.

Similarly, it takes longer to read data from physical memory than from the cache, and this can be exploited to know if some memory has been accessed recently. The infamous spectre and meltdown vulnerabilities uses this to know what branch was taken [3, 4].

In our case, timing garbage collection can reveal which branch was taken in an if statement if one branch produces more garbage than the other.

2.3 The Pedersen-Askarov Attack

The timing attack found by Pedersen and Askarov, which we will refer to as the *Pedersen-Askarov attack*, is possible due to some properties in garbage collectors. Only one of these properties are needed to be able to exploit them into a side-channel attack. We will focus on copy collectors, but the mark-and-sweep algorithm is also vulnerable [6].

2.3.1 High dependency in low context

During garbage collection, the number of bytes copied from the *from-space* to the *to-space* depends on how much memory is reachable before collection begins.

Figure 1 shows how this can be exploited. Data is allocated for the array `a` of `size1`. Then, depending on the secret variable `h`, memory is allocated and assigned to either variable `b` or `c`.

A reference to the previously allocated array `a` is also assigned to `d`, or `b`.

On line 13, c is assigned `null` and in the case $h \not> 0$ the newly allocated memory is now garbage.

```
 1  int[] a = new int[size1];
 2  int[] b = null;
 3  int[] c = null;
 4  int[] d = null;
 5  if (h > 0) {
 6      b = new int[size1];
 7      d = a;
 8  }
 9  else {
10      c = new int[size1];
11      b = a;
12  }
13  c = null;
14  long before = System.nanoTime();
15  //GC is triggered:
16  int[] x = new int[size2];
17  long after = System.nanoTime();
18  long diff = after - before;
```

Figure 1. Example Java program leaking one bit based on GC time. (Example taken from [6])

When allocating a new array at line 14, the GC threshold is reached and collection is started. Taken together, in the case $h > 0$, `diff` will be larger.

2.3.2 Low modification in high context

In addition to how garbage collecting sensitive information in a low context can be used to generate a side-channel, garbage collecting public information in a high context can also leak information.

In Figure 2, let's say that `size1+size2` $<$ *collectionThreshold* and `2·size1+size2` $>$ *collectionThreshold*. In the case of $h \not> 0$, the collection threshold will be reached at line 9, triggering the GC. In the case of $h > 0$, the GC will be triggered at line 5, meaning the heap will have been compacted already at line 9, and there will be no need for the GC to run again.

```
 1  int[] a = new int[size1];
 2  a = new int[size1];
 3  if (h > 0) {
 4      //GC is triggered:
 5      b = new int[size2];
 6      int[] b = null
 7  }
 8  long before = System.nanoTime();
 9  int[] x = new int[size2];
10  long after = System.nanoTime();
11  long diff = after - before;
```

Figure 2. Triggering GC in high context is also problematic.

```
 1  var cs : List[Int] = List.range(1,size1)
 2  var list : List[Int]
 3  if (h > 0) {
 4      list = 0 :: cs
 5  }
 6  else {
 7      list = 0 :: Nil
 8  }
 9  cs = Nil
10  var before = System.nanoTime()
11  var xs : List[Int] = List.range(1,size2)
12  var after = System.nanoTime()
13  var diff = after - before
```

Figure 3. Scala example: Depending on `h`, list could have a pointer to `cs`

2.3.3 Pointers from high to low

In Figure 3, `list` is either the list `[0]` or `0` prepended to the low level list `cs`. This means that even though we allocate one list node in both branches and assign them to `list`, in one branch it has a pointer to the first version of `cs`, and we can't throw it away when the GC is triggered at line 11.

Once again, the amount of live data depends on the secret value `h`.

2.3.4 Implications and Mitigation

The solution consists of making sure that garbage collection in a low context is time independent of high. One way to do this is to make garbage collection in low context completely independent of high.

The solution proposed by Pedersen and Askarov consists of segregating high and low level data. The memory is divided in two separate heaps, the high heap and the low heap.

When memory is allocated in low context, the "low GC" allocates memory on the low heap, and when memory is allocated in high context, the "high GC" allocates memory on the high heap.

This solves the problem presented in section 2.3.1. The problem in 2.3.2 is solved by never collecting the low heap in a high context. The problem presented in 2.3.3 is harder. If there are pointers from high to low, then the low GC will have to scan the high heap for pointers to low before it knows what data on the low heap is live.

Note that the problem is not that high context behavior depends on low. Low security level variables and data can be read in a high context, but the use of this information must not affect the low level.

The safest way to solve this problem is to simply not allow pointers to low level data to be kept when exiting the high context. This means banning pointers from the high heap to the low heap.

3 Project goal

This project aims to demonstrate the Pedersen-Askarov attack in a program using a simple garbage collector, and how changing the behavior of the garbage collector mitigates the attack.

We do this by implementing a simple copying garbage collector in C. The implementation is limited to a proof-of-concept, and not intended for real use. The garbage collector should be able to compact data reachable from a set of root pointers. The root pointers are supplied manually, rather than the collector scanning the stack. The heap size is hard-coded for simplicity, and does not grow.

This simple garbage collector will then be modified into a "hardened" collector with segregated heaps making it immune to the attack.

The collector will be used to show a demo of the Pedersen-Askarov attack, first on the basic copy collector and then on the hardened collector.

4 Implementation

Implementation of the GC was done in C, and all code is in one single file. The code is not object oriented, rather it's structured programming. The code is divided into functions that each have a single purpose so that no function gets too large.

4.1 Allocation

The two most important structs are `memheader` (Figure 4) and `gc_heap` (Figure 5).

A heap is initialized by allocating memory of the size `MAX_HEAPSIZE`. In our current implementation that is 1 GiB. A struct (Figure 5) holds pointers that mark the beginning, end, and break of the heap. The break marks the border between the allocated space and the free space.

Memory is allocated in "blocks". The function is called `galloc`, as in **g**arbage collected **alloc**ation.

```c
typedef struct header {
    struct header * next;
    unsigned int size;
    int security_level;
    void * forwarding;
} memheader;
```

Figure 4. Header for allocated blocks of memory

```c
typedef struct gc_heap {
    void* start;
    void* end;
    void* gbreak;
    int security_level;
} gc_heap;
```

Figure 5. Struct holding pointers to the heap

When memory is allocated, a header (Figure 4) is written before the allocated memory that holds the size of the allocated block, a pointer to where the next block will be inserted, and a field for a forwarding pointer to be used during collection. The `security_level` will be used by the hardened version of the collector.

When the space between the break and the end of the heap is too small, the heap must be garbage collected before more allocations can be made.

4.2 Algorithm

The algorithm can be divided into three important functions. The `collect` function, the `balloc` (block allocation) and the `scan_block` function. To keep track of the blocks that are to be scanned and copied, a stack is used. We call this stack the "scan stack".

4.2.1 Block allocation

`balloc` takes a pointer to a block's header and allocates a block of the same size on the new heap. It also sets the forwarding pointers of the blocks to point to each others memory regions (the memory after the block). The return value is a pointer to the *block* rather than to the actual memory like in `galloc`.

4.2.2 Collect

1. For each root pointer:
   a. call `balloc` on the corresponding block
   b. add the block to the scan stack
2. While the scan stack is not empty:
   a. remove a block from the stack
   b. scan it for pointers to other blocks using `scan_block`
   c. copy the contents of the block to the new location
3. For all blocks on the new stack: remove the pointer to the old block
4. Replace the root pointers with pointers to the corresponding new allocations

4.2.3 Scan block

1. For every 64 bit chunk of memory in the block:
   a. see if it can be interpreted as a pointer to an appropriate address
   b. If there is a no forwarding pointer:
      i. Call `balloc` on the block
      ii. Add the block to the scan stack
   c. replace the found pointer with the forwarding pointer

4.3 Structure

The `main` function demos the implementation by allocating data with `galloc` and invoking the Garbage Collector with `collect`, which mostly does calls to `scan_block` and `copy_block`.

4.4 Files

The implementation consists of one single file, containing a multitude of functions, all used to complete the goal of a garbage collection. The main function is `collect`. `collect` creates temporary stacks and pointers that are to be used during the computation. It then uses a helper function to scan blocks of memory for any references to other blocks of memory, that in the end should be copied and saved. Whilst a scanning is underway, it will search through a block of data, check for references, and if there are any, scan them too. When the scanning of a block is complete, the block is copied over to the new heap.

4.5 Test Results

To test the garbage collector, a simple program allocating a linked list was made. The main function of this program can be found in Appendix A.9.

The program declares a simple linked list of strings. Every list node has a pointer to it's string element, and a pointer to the next list node. A list is allocated holding the 5 strings: “hej” “du” “ditt” “lilla” “skräp”.

Figure 6 shows the log output of running the collect function with only the fourth list node as root pointer, and then printing the original list and last collected list on the new heap.

For each list node the address of the list, and the address and contents of the string is printed.

```
scanning block:        0×7fd0316a7088
found address:         0×7fd0316a7174
balloc:                0×7fcff16a6038
forwarding:            0×7fcff16a6050
current scanner:       0×7fd0316a7174
updated scanner:       0×7fcff16a6050
found address:         0×7fd0316a70c8
balloc:                0×7fcff16a6064
forwarding:            0×7fcff16a607c
current scanner:       0×7fd0316a70c8
updated scanner:       0×7fcff16a607c
scanning complete!

moving
0×7fd0316a7088 →       0×7fcff16a6010
moving complete!

scanning block:        0×7fd0316a70b0
found address:         0×7fd0316a71a0
balloc:                0×7fcff16a608c
forwarding:            0×7fcff16a60a4
current scanner:       0×7fd0316a71a0
updated scanner:       0×7fcff16a60a4
scanning complete!

moving
0×7fd0316a70b0 →       0×7fcff16a6064
moving complete!

scanning block:        0×7fd0316a7188
scanning complete!

moving
0×7fd0316a7188 →       0×7fcff16a608c
moving complete!

scanning block:        0×7fd0316a715c
scanning complete!

moving
0×7fd0316a715c →       0×7fcff16a6038
moving complete!

0×7fd0316a7028  0×7fd0316a70f0  hej
0×7fd0316a7050  0×7fd0316a711c  du
0×7fd0316a7078  0×7fd0316a7148  ditt
0×7fd0316a70a0  0×7fcff16a6050  lilla
0×7fcff16a607c  0×7fcff16a60a4  skräp
0×7fcff16a607c  0×7fcff16a60a4  skräp


0×7fcff16a6028  0×7fcff16a6050  lilla
0×7fcff16a607c  0×7fcff16a60a4  skräp
```

Figure 6. Output

```c
int main(int argc, char *argv[]) {
    int h;

    sscanf(argv[1], "%d\n", &h);

    int* a = galloc(SIZE1);
    int* b = NULL;
    int* c = NULL;
    int* d = NULL;

    if (h > 10) {
        b = galloc(SIZE1);
        d = a;
    }
    else {
        c = galloc(SIZE1);
        b = a;
    }
    c = NULL;
    clock_t before, after;
    before = clock();
    int* x = galloc(SIZE2);

    void* roots[] = {a,b,c,d,x};
    collect(5, roots);
    after = clock();
    long diff = after - before;

    printf("cycles: %d\n", diff);
}
```

Figure 7. The attack run via Main, written in C

5 Results

5.1 Porting the Pedersen-Askarov attack

To put the garbage collector to the test, the example in Figure 1 was ported to work with the new garbage collector. (Figure 7)

Running the code yielded the output visible in Figure 9. Figure 8 makes it extremely clear that there is a distinct difference depending on which branch was taken.

5.2 Hardening the collector

To mitigate the attack the implementation was changed to use one heap for each security level. The changes can be seen in Appendix B.

The new implementation supports multiple heaps. Running the garbage collector in a low context will only look at root pointers that are low. Pointers to blocks that are high are ignored. Keeping the security level in the block headers make this easier.

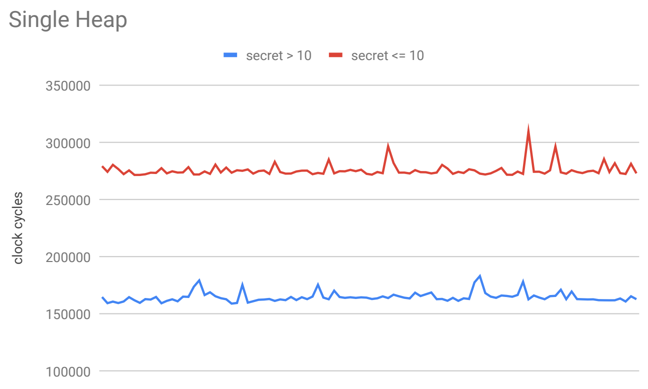

Figure 8. GC time measured in clock cycles for the two branches.

```
$ ./a.out 0
cycles: 156376
$ ./a.out 0
cycles: 155160
$ ./a.out 0
cycles: 153099
$ ./a.out 0
cycles: 152899
$ ./a.out 20
cycles: 257661
$ ./a.out 20
cycles: 260445
$ ./a.out 20
cycles: 269554
$ ./a.out 20
cycles: 254465
```

Figure 9. The first branch consistently takes longer to run than the second (The value 0 or 20 is what `h` is set to for that execution)

A global variable with a pointer to a heap struct (Figure 5) represents what level of security the current context is.

The `current_heap` was assigned to the corresponding heap when switching between low and high context. The new main can be seen in Figure 10.

The new heap is separated into two separate heaps. One for high level security, and one for low. This means that the garbage collector does not collect these at the same time In low execution context, only the low heap is collected.

The hardened garbage collector imposes the following restrictions:

A pointer from high data to low data is not allowed, because then the GC can't collect the low

```
int main(int argc, char *argv[]) {

    int h;

    sscanf(argv[1], "%d\n", &h);

    int* a = galloc(SIZE1);
    int* b = NULL;
    int* c = NULL;
    int* d = NULL;

    current_heap = &highHeap;
    if (h > 10) {
        b = galloc(SIZE1);
        d = a;
    }
    else {
        c = galloc(SIZE1);
        b = a;
    }
    current_heap = &lowHeap;
    c = NULL;
    clock_t before, after;
    before = clock();
    int* x = galloc(SIZE2);

    void* roots[] = {a,b,c,d,x};
    collect(5, roots);
    after = clock();
    long diff = after - before;

    printf("cycles: %d\n", diff);
}
```

Figure 10. `current_heap` determines whether *PC* = *high* or *PC* = *low*

heap without first sweeping the high heap and finding pointers to the low heap. If this was allowed, collecting the low heap would depend on high level data, which we can't allow.

Pointers from low data to high data is not an issue for the GC, so there is no reason to disallow them for memory management to work properly.

The collection itself does things a bit different depending on the security level. When doing a low garbage collection, only the low heap is looked at, and from the root pointers, the pointed to data is copied into a new memory allocation. Any pointer to high data is copied with the same address, so the location of the pointer itself is moved to the new heap, but where it points to in the high data remains the same.

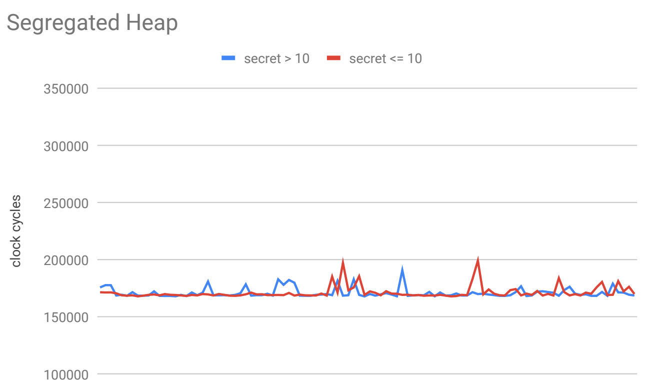

Figure 11. GC time measured in clock cycles for the two branches.

```
$ ./a.out 0
cycles: 160750
$ ./a.out 0
cycles: 161643
$ ./a.out 0
cycles: 167376
$ ./a.out 0
cycles: 165160
$ ./a.out 20
cycles: 169385
$ ./a.out 20
cycles: 169077
$ ./a.out 20
cycles: 166916
$ ./a.out 20
cycles: 163836
```

Figure 12. In the new version, there's no noticeable difference in timing.

When collecting the high data, both heaps are traversed to find which objects on the high heap to save. Objects on the low heap are not collected, but if they contain pointers to high objects, they are rewritten so that they point to the new data.

5.3 Attack on hardened garbage collector

With the new collector in place, and the main updated to change `current_heap` when switching execution context, we run the code again. In Figure 12 and 11 it's clear that the differences seen previously is no longer noticeable. There is no measurable difference in GC time depending on branch.

6 Discussion

By implementing a simple copying garbage collector, we showed in practice, that a simple collection algorithm is vulnerable to a timing attack breaking guarantees otherwise promised by information flow control. We change the implementation and show that the attack is no longer possible on the hardened version of the collector.

6.1 Allowing pointers from high to low

The limitation that high level data can't refer to low level data is a tough one. Finding a better solution to this problem would be a good topic for future research.

There are probably many ways to solve this problem, all with it's own drawbacks. A solution worth exploring could however be to mark the low heap when entering high context, and then keep that heap segment until high context is entered the next time.

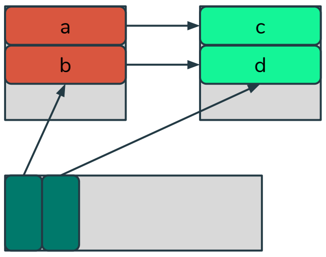

Figure 13. There are pointers from high to *c* and *d*. As far as low is concerned, *c* is garbage.

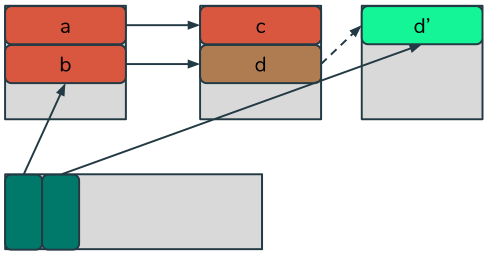

Figure 14. The low heap is garbage collected. *c* is only reachable from high. *d′* is now in the new heap, a forwarding pointer is left pointing from *d* to *d′*.

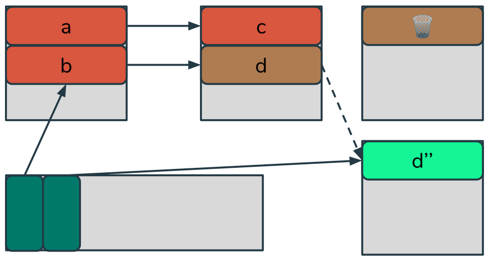

Figure 15. The low heap is garbage collected again. The forwarding pointer is updated so that it points from *d* to *d″*. The space in the previous low heap can now be reclaimed.

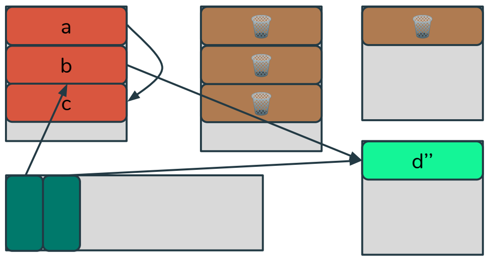

Figure 16. Finally, high context is entered and it can replace the pointer to *d* with *d″* and put *c* on the high heap.

In the example shown in Figure 13-16, this flow is shown:

- *c* and *d* is allocated on the low heap.
- High context is entered, and *a* and *b* are allocated with pointers to *c* and *d*.
- Low context is entered, the reference to *c* is dropped and GC is triggered, copying *d* to the new location *d′*. The GC leaves a forwarding pointer from *d* to *d′* and a pointer from *d′* to the original *d*.
- GC is triggered again and *d′* is collected to new location *d″*. The pointer to the original *d* is used to find it and the forwarding pointer is updated so that it points to *d″*.
- High context is entered and before anything else is done, *c* is collected to the high heap since low doesn't need it anymore. Pointers to *d* is updated so that they point to *d″* instead.

- When low context is entered again. The low GC can now be sure that there's nothing important on the heap where c and d originally resided.

The advantage of this solution is that the space leak is bounded by the size of the first low heap. There is however a flaw in the solution. The idea is that the heap where d' resides can be reclaimed directly once the low GC is finished, but what if high context needs an object that is dropped on that heap?

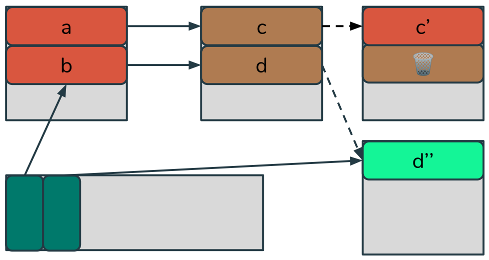

Figure 17. Reclaiming the second low heap would overwrite c'

Consider Figure 19. If the low GC now overwrites c', that object will be lost. If all data is immutable, then the original c will suffice, but the bookkeeping required to remove the forwarding pointer from c to c' is unnecessarily inefficient.

Juggling different versions of objects simply because high data might need them is asking for trouble. Leaving c and d in their original place and leaving it to the high GC to collect them would've been easier. So why not do just that?

The reason the low heap is not allowed to be collected in high is that it would affect at what point in the future the low GC is triggered. When the low GC is triggered, or for how long it runs must not be affected by high level behavior. If the high GC always takes control of the low heap when high context is entered, and the low GC starts on a new heap, this problem is solved.

The new strategy divides memory into three parts, the high heap, which only the high GC is allowed to read or modify; the low heap, where the low GC allocates memory and reclaims space as needed; and an intermediate heap, which low context is allowed to read, but not manage.

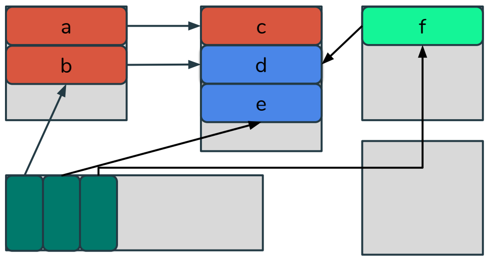

Figure 18. The old low heap is now controlled by the high GC. Low allocates f on a new heap, it can read the old heap, but can't move anything on it, or use it for new allocations.

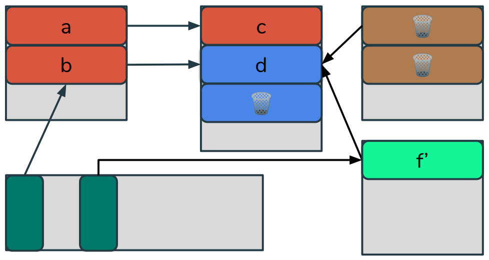

Figure 19. The program progresses, and f is collected to a new low heap. The low heap with only garbage can be reclaimed. e on the first low heap is garbage now, but we have to wait for the high GC to collect it.

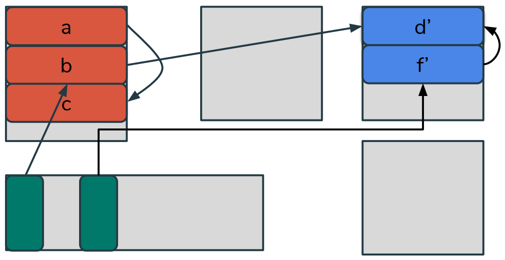

Figure 20. High context is entered, and the high GC collects all low objects and puts them on a new heap.

Every time context goes from low to high, the low heap is collected onto the intermediate heap. When low context is reentered, new allocations are made on a fresh empty heap. Once a threshold is reached, the low GC collects the low heap, reclaiming space, but doesn't touch the intermediate heap. When the high heap is entered, low and intermediate heap can be collected and compacted into a new intermediate heap.

The strategy could be improved by analyzing statically what objects high context can make references to in the future and only allocating those on the intermediate heap. A drawback of this approach compared to simply not allowing pointers from high to low, is of course that some garbage will have to wait before it can be reclaimed. This might not be a problem in practice though. It has been shown in practice that "most objects die young", also known as the *weak generational hypothesis* [2, 8]. Assuming that the hypothesis is true, only collecting the low heap in low context and letting the rest wait for high context might be enough. Whether this strategy is enough to prevent information leakage remains to be proven.

7 Conclusion

In this project we managed to implement a simple copying garbage collector and show that it is vulnerable to the attack presented in Pedersen and Askarov's paper [6]. The attack is mitigated by segregating the memory into heaps corresponding to security level. Running the attack on the hardened garbage collector shows that it no longer leaks information. We also discuss solutions to the problem of pointers from high to low.

Exploiting the behavior of garbage collectors to extract information is a new topic in information flow control, and poses very interesting challenges to further research.

References

[1] Pablo Buiras and Alejandro Russo. 2013. Lazy programs leak secrets. In *Nordic Conference on Secure IT Systems*. Springer, 116–122.

[2] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC.

[3] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[4] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[5] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[6] Mathias V Pedersen and Aslan Askarov. 2017. From trash to treasure: timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 693–709.

[7] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[8] David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan notices*, Vol. 19. ACM, 157–167.

[9] Steve Zdancewic and Andrew C Myers. 2003. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* IEEE, 29–43.

A Source code

The following are code snippets of most of the functions.

A.1 galloc

```
/* Works like malloc, but for our garbage collected heap
 * Returns a pointer to the newly allocated data.
 */
void * galloc(int size){
    if (current_heap->end == NULL) {

        current_heap->start = malloc(MAX_HEAPSIZE);
        current_heap->gbreak = current_heap->start;
        current_heap->end = (char*) (current_heap->start) + MAX_HEAPSIZE;
    }

    if ((ptrdiff(current_heap->gbreak, current_heap->start) + sizeof(memheader) + size) > MAX_HEAPSIZE) {
        printf("Something went wrong\n");
        //collect();
        return NULL;
    }

    memheader* header = current_heap->gbreak;
    void* block = current_heap->gbreak + sizeof(memheader);
    current_heap->gbreak += sizeof(memheader) + size;
    //printf("Header: %p\nBlock: %p\nHeapened: %p\n", header, block, heapend);

    header->next = current_heap->gbreak;
    header->size = size;
    header->security_level = current_heap->security_level;
    header->forwarding = NULL;

    return block;
}
```

A.2 push_block

```
/* Push block onto stack and return block (new stack)
*/
memheader* push_block(memheader* stack,
    memheader* block) {
    block->next = stack;
    return block;
}
```

A.3 pop_block

```
memheader* pop_block(memheader* stack) {
    memheader* top_block = stack;
    stack = stack→next;
    top_block→next = ((void*) top_block) + sizeof(memheader) + top_block→size;
    if (ptrdiff(top_block→next, top_block+1) ≠ top_block→size) {
        fprintf(stderr, "\nWARNING:
            top_block→next incorrect\n");
    }
    return stack;
}
```

A.4 copy_block

```
/* Takes a pointer to the new block
 * and copies the old data to the new location
*/
void copy_block(memheader* old_block) {
    void* old_data = old_block+1;
    memheader* new_block = forw_header(old_block);
    void* new_data = old_block→forwarding;

    fprintf(stderr, "moving\n%p →\t%p\n",old_block, new_block);

    if (forw_header(new_block) ≠ old_block) {
        fprintf(stderr, "WARNING: forw_header(new_block) ≠ old_block\nnew: %p\nold: %p\n", new_block, old_block);
    }

    memcpy(new_data, old_data, old_block→size);

    fprintf(stderr, "moving complete!\n\n" );
}
```

A.5 balloc

```
/* Allocate memory for a block to be copied to a new location
 * Takes a block and allocates a new block of the same size
 * The blocks are coupled with forwarding pointers so that
 * the new block can be found from the old and vice versa.
*/
memheader* balloc(memheader* old_block) {
    memheader* new_data = galloc(old_block→size);
    memheader* new_block = new_data-1;
    old_block→forwarding = new_data;
    new_block→forwarding = old_block+1;
    new_block→security_level = old_block→security_level;
    return new_block;
}
```

A.6 addr_in_heap

```
bool addr_in_heap(void* address) {
    if (highHeap.start < address) {
        if (address < highHeap.end) {
            return true;
        }
    }
    if (lowHeap.start < address) {
        if (address < lowHeap.end) {
            return true;
        }
    }
    return false;
}
```

A.7 scan_block

```
memheader* scan_block(memheader* block, memheader* scan_stack) {
    fprintf(stderr, "scanning block:\t\t%p\n", block);
    memheader** scanner = (memheader**) (block+1);
    while((memheader*) scanner < block->next) {
        if (addr_in_heap(*scanner)) {
            fprintf(stderr, "found address:\t\t%p\n", *scanner);
            memheader* found = (*scanner)-1;

            if (found->security_level == current_heap->security_level) {
                if (found->forwarding == NULL) {
                    scan_stack = push_block(scan_stack, balloc(found));
                    fprintf(stderr,"balloc:\t\t\t%p\n", scan_stack);
                } else if (forw_header(forw_header(found)) == found) {
                    fprintf(stderr, "address already forwarded");
                } else {
                    scanner++;
                    continue;
                }
                fprintf(stderr,"forwarding:\t\t%p\n", found->forwarding);
                fprintf(stderr,"current scanner:\t%p\n", *scanner);
                *scanner = found->forwarding;
                fprintf(stderr,"updated scanner:\t%p\n", *scanner);
            }
        }
        scanner++;
    }
    fprintf(stderr, "scanning complete!\n\n", block);
    return scan_stack;
}
```

A.8 collect

```
void collect(int rootlen, void** roots) {

    memheader *scan_stack, *block;
    scan_stack = NULL;

    empty.security_level = current_heap→security_level;

    gc_heap tmp = *current_heap;
    *current_heap = empty;
    empty = tmp;

    old_heap = &empty;

    for (int i = 0; i < rootlen; i++) {
        if (roots[i] ≠ NULL) {
            block = roots[i];
            //The header is located before the data.
            block--;
            //scan_block will not balloc the initial block, only found ones.
            //push a newly allocated block on the stack:
            if (block→security_level ⩽ current_heap→security_level)
                scan_stack = push_block(scan_stack, balloc(block));
        }
    }

    while (scan_stack ≠ NULL) {

        //forw_header gives back the old block
        block = forw_header(scan_stack);
        scan_stack = pop_block(scan_stack);

        scan_stack = scan_block(block,scan_stack);

        copy_block(block);
    }

    //reset all forwarding pointers
    block = current_heap→start;
    while (block < (memheader*) current_heap→gbreak) {
        block→forwarding = NULL;
        block = block→next;
    }

    //give back the new roots
    for (int i = 0; i < rootlen; i++) {
        if (roots[i] ≠ NULL) {
            block = roots[i];
            block--;
            roots[i] = block→forwarding;
        }
    }

    //reset the empty heap
    empty.gbreak = empty.start;
    empty.security_level = -1;

}
```

A.9 Linked List Demo

```c
int main(int argc, char *argv[]) {

    struct List {
        char* head;
        struct List * tail;
    };

    struct List * list = galloc(sizeof(struct List));
    list->tail = galloc(sizeof(struct List));
    list->tail->tail = galloc(sizeof(struct List));
    list->tail->tail->tail = galloc(sizeof(struct List));
    list->tail->tail->tail->tail = galloc(sizeof(struct List));
    list->tail->tail->tail->tail->tail = NULL;

    char* str = galloc(20);
    strcpy(str, "hej");
    list->head = str;

    str = galloc(20);
    strcpy(str, "du");
    list->tail->head = str;

    str = galloc(20);
    strcpy(str, "ditt");
    list->tail->tail->head = str;

    str = galloc(20);
    strcpy(str, "lilla");
    list->tail->tail->tail->head = str;

    str = galloc(20);
    strcpy(str, "skräp");
    list->tail->tail->tail->tail->head = str;

    memheader* block = (void*) list->tail->tail->tail;
    block--;
    collect(block);

    while(list!=NULL) {
        printf("%p\t%p\t%s\n", list, list->head, list->head);
        list=list->tail;
    }

    printf("%s\n\n" );

    list=block->forwarding;

    while(list!=NULL) {
        printf("%p\t%p\t%s\n", list, list->head, list->head);
        list=list->tail;
    }
  }
```

A.10 Timing attack on vulnerable garbage collector

```c
int main(int argc, char *argv[]) {

    int h;

    sscanf(argv[1], "%d\n", &h);

    int* a = galloc(SIZE1);
    int* b = NULL;
    int* c = NULL;
    int* d = NULL;

    if (h > 10) {
        b = galloc(SIZE1);
        d = a;
    }
    else {
        c = galloc(SIZE1);
        b = a;
    }
    c = NULL;
    clock_t before, after;
    before = clock();
    int* x = galloc(SIZE2);

    void* roots[] = {a,b,c,d,x};
    collect(5, roots);
    after = clock();
    long diff = after - before;

    printf("cycles: %d\n", diff);
}
```

A.11 Timing attack on hardened garbage collector

```c
int main(int argc, char *argv[]) {

    int h;

    sscanf(argv[1], "%d\n", &h);

    int* a = galloc(SIZE1);
    int* b = NULL;
    int* c = NULL;
    int* d = NULL;

    current_heap = &highHeap;
    if (h > 10) {
        b = galloc(SIZE1);
        d = a;
    }
    else {
        c = galloc(SIZE1);
        b = a;
    }
    current_heap = &lowHeap;
    c = NULL;
    clock_t before, after;
    before = clock();
    int* x = galloc(SIZE2);

    void* roots[] = {a,b,c,d,x};
    collect(5, roots);
    after = clock();
    long diff = after - before;

    printf("cycles: %d\n", diff);
}
```

B Diff between vulnerable and hardened

B.1 Top level declarations

```
index 44fccf0..c02a0ec 100644
--- b/lek.c
+++ a/lek.c
@@ -7,19 +7,28 @@
 typedef struct header {
     struct header * next;
     unsigned int size;
+    int security_level;
     void * forwarding;
 } memheader;

 #define MAX_HEAPSIZE (1024*1024*1024)

+#define LOW 0
+#define HIGH 1
+
 typedef struct gc_heap {
     void* start;
     void* end;
     void* gbreak;
+    int security_level;
 } gc_heap;

-gc_heap heap;
-gc_heap empty;
+gc_heap highHeap = {NULL, NULL, NULL, HIGH};
+gc_heap lowHeap = {NULL, NULL, NULL, LOW};
+gc_heap empty = {NULL, NULL, NULL, -1};
+
+gc_heap* current_heap = &lowHeap;
+gc_heap* old_heap;
```

B.2 galloc

```
index e476d4c..44fccf0 100644
--- b/lek.c
+++ a/lek.c
@@ -33,26 +33,27 @@{
 void * galloc(int size){
-    if (heap.end == NULL) {
+    if (current_heap->end == NULL) {

-        heap.start = malloc(MAX_HEAPSIZE);
-        heap.gbreak = heap.start;
-        heap.end = (char*) (heap.start) + MAX_HEAPSIZE;
+        current_heap->start = malloc(MAX_HEAPSIZE);
+        current_heap->gbreak = current_heap->start;
+        current_heap->end = (char*) (current_heap->start) + MAX_HEAPSIZE;
     }

-    if ((ptrdiff(heap.gbreak, heap.start) + sizeof(memheader) + size) > MAX_HEAPSIZE) {
+    if ((ptrdiff(current_heap->gbreak, current_heap->start) + sizeof(memheader) + size) > MAX_HEAPSIZE) {
        printf("ajabaja\n");
        //collect();
        return NULL;
    }

-    memheader* header = heap.gbreak;
-    void* block = heap.gbreak + sizeof(memheader);
-    heap.gbreak += sizeof(memheader) + size;
+    memheader* header = current_heap->gbreak;
+    void* block = current_heap->gbreak + sizeof(memheader);
+    current_heap->gbreak += sizeof(memheader) + size;
     //printf("Header: %p\nBlock: %p\nHeapened: %p\n", header, block, heapend);

-    header->next = heap.gbreak;
+    header->next = current_heap->gbreak;
     header->size = size;
+    header->security_level = current_heap->security_level;
     header->forwarding = NULL;

     return block;
```

B.3 balloc

```
index c2f1eb2..e476d4c 100644
--- b/lek.c
+++ a/lek.c
@@ -119,6 +119,7 @@ memheader* balloc(memheader* old_block) {
     memheader* new_block = new_data-1;
     old_block->forwarding = new_data;
     new_block->forwarding = old_block+1;
+    new_block->security_level = old_block->security_level;
     return new_block;
 }
```

B.4 scan block

```
index bae306e..c2f1eb2 100644
--- b/lek.c
+++ a/lek.c
@@ -122,6 +122,20 @@ memheader* balloc(memheader* old_block) {
     return new_block;
 }

+bool addr_in_heap(void* address) {
+    if (highHeap.start < address) {
+        if (address < highHeap.end) {
+            return true;
+        }
+    }
+    if (lowHeap.start < address) {
+        if (address < lowHeap.end) {
+            return true;
+        }
+    }
+    return false;
+}
```

```
+
@@ -129,10 +143,11 @@ memheader* scan_block(memheader* block, memheader* scan_stack) {
     fprintf(stderr, "scanning block:\t\t%p\n", block);
     memheader** scanner = (memheader**) (block+1);
     while((memheader*) scanner < block->next) {
-        if ((memheader*) empty.start < *scanner) {
-            if (*scanner < (memheader*) empty.end) {
-                fprintf(stderr, "found address:\t\t%p\n", *scanner);
-                memheader* found = (*scanner)-1;
+        if (addr_in_heap(*scanner)) {
+            fprintf(stderr, "found address:\t\t%p\n", *scanner);
+            memheader* found = (*scanner)-1;
+
+            if (found->security_level == current_heap->security_level) {
                 if (found->forwarding == NULL) {
                     scan_stack = push_block(scan_stack, balloc(found));
                     fprintf(stderr,"balloc:\t\t\t%p\n", scan_stack);
```

B.5 collect

```
index 53c4d4a..bae306e 100644
--- b/lek.c
+++ a/lek.c
@@ -162,10 +162,14 @@ void collect(int rootlen, void** roots) {
     memheader *scan_stack, *block;
     scan_stack = NULL;

-    gc_heap tmp = heap;
-    heap = empty;
+    empty.security_level = current_heap->security_level;
+
+    gc_heap tmp = *current_heap;
+    *current_heap = empty;
     empty = tmp;

+    old_heap = &empty;
+
     for (int i = 0; i < rootlen; i++) {
         if (roots[i] != NULL) {
             block = roots[i];
@@ -173,7 +177,8 @@ void collect(int rootlen, void** roots) {
             block--;
             //scan_block will not balloc the initial block, only found ones.
             //push a newly allocated block on the stack:
-            scan_stack = push_block(scan_stack, balloc(block));
+            if (block->security_level <= current_heap->security_level)
+                scan_stack = push_block(scan_stack, balloc(block));
         }
     }

@@ -189,12 +194,13 @@ void collect(int rootlen, void** roots) {
     }

     //reset all forwarding pointers
-    block = heap.start;
-    while (block < (memheader*) heap.gbreak) {
+    block = current_heap->start;
+    while (block < (memheader*) current_heap->gbreak) {
         block->forwarding = NULL;
         block = block->next;
     }

+    //give back the new roots
     for (int i = 0; i < rootlen; i++) {
         if (roots[i] != NULL) {
             block = roots[i];
@@ -203,6 +209,10 @@ void collect(int rootlen, void** roots) {
         }
     }

+    //reset the empty heap
+    empty.gbreak = empty.start;
+    empty.security_level = -1;
+
 }
```

B.6 main

```
index ff28c50..53c4d4a 100644
--- b/lek.c
+++ a/lek.c
@@ -224,6 +224,7 @@ int main(int argc, char *argv[]) {
     int* c = NULL;
     int* d = NULL;

+    current_heap = &highHeap;
     if (h > 10) {
         b = galloc(SIZE1);
         d = a;
@@ -232,6 +233,7 @@ int main(int argc, char *argv[]) {
         c = galloc(SIZE1);
         b = a;
     }
+    current_heap = &lowHeap;
     c = NULL;
     clock_t before, after;
     before = clock();
```