Rust and the importance of memory safe systems programming languages

MARC ROBERT WONG

1 SUMMARY

As technology has become more and more integrated in our lives, it is becoming more and more important to be able to trust in the security of the systems we use every day. Just like you would not trust a bank with no vault or want to store your mail in an unlocked room, it is of utmost importance to ensure that our technology is also as secure as possible.

In this report, I will attempt to demonstrate the importance of memory safe systems programming languages by showing that memory safe languages are the best defense against the most critical security exploits.

First, I will begin by introducing memory safety and showing how memory safety and security are intimately linked. Next, I will examine in detail some well-known examples of memory related exploits in order to better understand how to defend against such exploits. Then, I will examine the status quo of defenses against such exploits at the platform level, runtime level and language level. Finally, I will explore how Rust is a unique systems programming language because it guarantees both memory safety and full control.

Through this research, I have determined that bringing memory safety to systems programming languages is the best defense against the most important security exploits. Although there is still merit to platform level and runtime level defenses, these will inevitably remain in a constant state of war between attacker and defender because they do not address the fundamental problem of being able to write insecure programs. Rust is an example of such a memory safe programming language that enforces safety and allows for full control and will therefore be instrumental in creating a new class of exploit resistant programs.

I hope that this report will demonstrate that it is very difficult to write vulnerability free programs in memory unsafe languages like C and C++ and encourage more of a move towards modernizing the systems programming landscape.

2 INTRODUCTION

2.1 What is memory safety?

In general, to say that a program or language is "memory safe" means that this program or language guarantees to never encounter any memory related errors such as buffer overflows, dangling pointers, or double frees. While it is possible to define memory safety as a long list of specific memory errors that it should prevent, this is not very intuitive. In "SoK: An Eternal War on Memory", the authors group these errors by defining memory safety as preventing all spatial and temporal memory errors. A memory error is spatial if it is an error because of reading or writing to a specific memory address that should not be valid. A memory error is temporal if it is an error because of, as the name would indicate, reading or writing to a memory address that should no longer be valid due to a timing issue. While this definition of memory safety may seem vague for now, these concepts of spatial and temporal memory errors will continue to appear throughout the remainder of this paper and will hopefully become more clear.

2.2 Why is memory safety important?

The next question that should be on your mind is asking yourself *why* it is important to ensure memory safety. Beyond avoiding logical errors or crashes, it is imperative to ensure memory safety because of the security risks that take advantage of the widespread lack of memory safety that exists in many execution contexts.

Although it is not immediately clear why memory safety could lead to a whole class of security vulnerabilities, this becomes more apparent if we take a step back and think more abstractly. Any computer program will have some elements of data and some elements of control. For example, in the context of a banking program, the data of this program would refer to the elements of the program that keep track of the bank account numbers, their balances, PINs, etc. The elements of control of this program refer to the logical backbone of the program that ensures that withdrawals are only possible after being authenticated via PIN, that a bank balance can never be negative, etc. If we continue examining this example, there will never be any issues assuming that the elements of control of the bank are properly implemented and they cannot be changed by anyone else.

However, the issue that arises in computer programs if there is not guaranteed memory safety is that the data and control of programs are mixed. This is an important design flaw that allows for non-memory safe programs to be vulnerable to what is known as control flow hijacking, where the attacker is able to take over control of a program by arbitrarily rerouting its control flow for nefarious purposes. While in other systems such as the telephone, the separation of data and control is possible, in most cases the best a computer program can do is to ensure memory safety in order to avoid a large amount of control flow hijacking attacks. In the next section, we will explore in detail a sampling of a few quintessential control flow hijacking attacks that occur because of a lack of memory safety.

3 MEMORY RELATED CONTROL FLOW HIJACKING ATTACKS

In order to explore concrete examples of control flow hijacking attacks that exploit a lack of memory safety, we can examine a few examples of quintessential security vulnerabilities in C, a language that is notorious for lacking any memory safety. These examples also demonstrate how easy it is to inadvertently write code with critical vulnerabilities. Finally, we will conclude this section by showing how memory safety errors are very prevalent in "real world" software, even today.

3.1 Buffer overflow: spatial

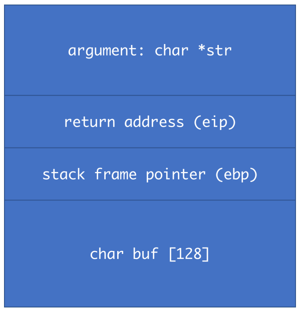
Buffer overflow attacks take advantage of the classic layout of a function stack in order to hijack the control flow of a program. In particular, the canonical buffer overflow exploit will figure out a way to overwrite past the end of a buffer allocated on the stack in order to overwrite the return address of the stack frame. This allows the attacker to control which set of instructions to execute next, thus hijacking the control flow of the program.

Let's consider the following C code:

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
}
```

In this function, our stack frame looks as follows:

Fig. 1. Diagram of stack frame layout



In this code, since there is no bounds checking in `strcpy` or in the C language in general, if the string `str` is longer than 128 bytes long, we will overflow the buffer and be able overwrite the stack frame pointer and return address. In this case, the simplest way to concretely exploit this vulnerability is to place some arbitrary code (ie. code that opens a shell) in the `buf` variable, then modify the return address to point to the beginning of `buf`.

The `buf` variable would then look something like:

```
0                      144              bytes
[--SHELLCODE--         ADDRESS_OF_BUF]  content
```

While this example is quite simple, the same idea of buffer overflows appear in many other exploits.

For example, even in a seemingly worst-case scenario of only being able to overflow a single byte, it is possible to successfully construct a control flow hijacking exploit.

Let's consider the following C code:

```
#include <stdio.h>

func(char *sm)
{
  char buffer[256];
  int i;
  for(i=0;i<=256;i++)
  buffer[i]=sm[i];
}

main(int argc, char *argv[])
{
  if (argc < 2) {
    printf("missing args\n");
    exit(-1);
  }

  func(argv[1]);
}
```

The error in this code is very subtle: instead of using `i<256`, the for loop in `func` has `i<=256`.

If we go back to our stack frame layout diagram, we can see that the only thing we will be able to modify in a 1 byte overflow is the last byte of the saved stack frame pointer `ebp`. This seems relatively innocuous, but if we consider the `CALL`, `LEAVE`, and `RET` procedures that occur during a function call and return, we can see why this will lead to an exploitable vulnerability. Recall that in a typical computer architecture, the stack grows down, `esp` refers to the top (low memory address) of the stack, `ebp` points to the current stack frame starting at locals and `eip` points to the current instruction.

`CALL foo()`:

- Push arguments onto the stack
- Push `eip` onto stack (saved return address)
- Push `ebp` onto stack (saved stack frame)
- Set `ebp` (start of new stack frame) = `esp` (top of old stack frame)
- Decrement `esp` in order to push stack variables

`LEAVE foo()`:

- Reverse the `CALL` procedure
- Shrink stack frame by setting `esp` (top of stack frame) = `ebp` (bottom of stack frame)
- Restore `ebp` to the saved stack frame by popping the stack

`RET foo()`:

- Restore `eip` to the saved return address by popping the stack

Our goal in this exploit is to point `esp` to a memory address containing the address of our exploit code such that when the `eip` is restored by popping the stack starting from `esp`, we will set the current instruction `eip` to point to the start of our exploit code.

The whole exploit can be summarized as follows:

- Execute `func()`: overflow the buffer by one byte and change the value of the last byte of the saved `ebp` in `func`.
- `LEAVE func()`
  - Shrink stack frame by setting `esp` (top of stack frame) = `ebp` (bottom of stack frame)
  - Restore `ebp` to the saved stack frame we modified by popping the stack.
- `RET func()`: restore `eip` to the saved return address by popping the stack
- We are now back in `main()`. When we `LEAVE main()`:
  - Shrink stack frame by setting `esp` (top of stack frame) = `ebp` (modified by exploit)
  - Restore `ebp` to the saved stack frame by popping the stack.
- `RET main()`: restore `eip` to the saved return address by popping the stack starting from the `esp` which we modified

In this exploit, our buffer would look something like this:

```
[no ops][EXPLOIT_CODE][&EXPLOIT_CODE][EBP_ALTERING_BYTE]
```

where we set `EBP_ALTERING_BYTE` such that the saved `ebp` has value of the address of the start of `&EXPLOIT_CODE` in our buffer - `0x4` (in order to account for popping `ebp` when we `LEAVE main()`). This exploit has an important caveat: if the caller's `ebp` is not located right above the destination buffer, then we will not be able to modify its last byte to point to the correct memory address.

Of course, these two buffer overflows are by no means the only exploits that involve overflowing some data structure to overwrite a memory location that controls the flow of the program. Other examples include buffer overflows that target exception handlers or function pointers, heap overflows which are very similar to buffer overflows similar but instead of targeting stack variables, instead attempt to overwrite control flow memory on the heap like virtual tables in C++.

There are also a multitude of exploits that use some sort of buffer overflow as the underlying exploit such as integer overflows where a bounds checking fails because of a large positive signed integer overflowing into a small negative integer or even a format string exploit which takes advantage of the internal stack pointer in format string functions like `printf` and the `%n` parameter which allows writing to addresses on the stack.

All of these exploits are examples of spatial memory errors because in some way or another, they all attempt to overwrite memory at a location that is critical to controlling the flow of the program which should be protected in a memory safe program as these regions are not within the valid range of memory regions.

3.2 Dangling pointer: temporal

Unlike the buffer overflow examples which are spatial memory errors, dangling pointer exploits and related errors are temporal memory errors. A "dangling pointer" occurs when a pointer to a freed object remains accessible.

As we will see through the examples in this section, the exploits occur because of valid memory functions like `free()` that are used in invalid combinations like freeing the same object twice. These exploits rely on knowing details of the specific memory allocation strategies of the underlying system, so I will attempt to explain a well-known exploit in the heap allocator included in the GNU C Library as well as a more general explanation of the use-after-free exploit present in many web browsers.

3.2.1 *Traditional double-free exploit: unlink() example.*

The `unlink()` exploit is a historical exploit in the GNU C library glibc that has since been patched that relies on a temporal error of freeing the same memory address twice. Let's consider a simplified demonstration of this exploit:

```
int func(char *arg) {
  char *p;
  char *q;
  p = malloc(500);
  q = malloc(500);
  free(p);
  free(q);
  p = malloc(1024);
  strncpy(p, arg, 1024);
  free(q);
}
```

In this code, we allocate two chunks "p" and "q" of size 500 bytes each. Then, we free both of these chunks, but importantly leave the pointers dangling to now invalid memory. If we then allocate another chunk of size 1024 bytes, we will re-use the same heap memory that was previously occupied by the two chunks "p" and "q". This means that we can construct malicious chunks of memory at the address previously occupied by "q". We can also note that "q" is freed once again after we allocate this memory.

In order to see why this can lead to an control flow exploit, we need to consider that in the glibc allocator, memory is stored in "chunks" in a doubly linked list sorted by size. Therefore, when a chunk is freed, the allocator will attempt to coalesce multiple adjacent free chunks by creating one larger chunk and removing this chunk from the doubly link list it was initially assigned to.

This is where the `unlink()` macro comes in:

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {       \
  FD = P->fd;                     \
  BK = P->bk;                     \
  FD->bk = BK;                    \
  BK->fd = FD;                    \
}
```

Equivalently, we have:

```
FD = *P + 8;
BK = *P + 12;
FD + 12 = BK;
BK + 8 = FD;
```

In particular, within our memory that we copy into "p" from `arg`, we will create two fake chunks of memory.

The first chunk will start at "q" and will contain the payload of arbitrary code we want to execute, and have a forward pointer (`fd`) to the second chunk.

The second chunk will start after the end of the first chunk and have forward pointer `fd` set to be the address we want to overwrite - 12 (ie. the return address of a function) and backwards pointer `bk` set to be the address of our payload in the first chunk. We also need to set the second chunk to have be a free chunk so that we will remove it from the doubly linked list when we free the first chunk.

Thus, when we call `free(q)` for the second time, we will indeed call the `unlink()` macro on the second chunk in an attempt to coalesce the two freed chunks. Therefore, we will set `FD + 12 = BK` or equivalently overwrite the address we gave (for example, return address of a function) with the address of our payload. An astute reader may notice that we will also overwrite some memory within our payload with some valid memory address. However, this will most likely not be valid assembly op codes, so we need to craft a special payload that includes a short jump instruction in order to "jump over" these invalid instructions and continue with the rest of our payload.

3.2.2 *Use-after-free.*

The main idea behind use after free is as follows. First, a valid object is allocated and freed. Next, we assume that there is a dangling pointer to this object that remains accessible. At this point, it is possible to allocate a new object that will now be placed in the same memory space as the first object. This object will be crafted maliciously, for example setting a function pointer to some arbitrary code. Finally, the original object is re-used (hence the name use-after-free) but since it now points to a malicious object, the arbitrary code will run.

Use-after-free vulnerabilities are very common in web browsers (specifically, the web browser engines) and difficult to detect because as mentioned before, they are a temporal memory issue that are much more subtle to detect than obviously incorrect behavior like writing data to invalid memory addresses. In addition, memory management is often difficult to understand in static analysis because the free and use of an object could be quite separated and occur because of complicated reasons.

In the "DangNull" paper that presents a runtime system of detecting use-after-free and double-free, the authors note that use-after-free is a very common exploit. For example, from 2011-2013, 680 of the total 929 total security vulnerabilities in Chromium were use-after-free vulnerabilities. Of those, 13 were considered to be of critical severity and 582 were considered to be of high severity.

3.3 Memory exploits in the real world

Although it may seem from the examples that we have explored so far that memory safety errors that lead to serious security vulnerabilities are few and far between and only occur in these specially crafted scenarios that would never really happen in practice, this is far from the truth. A large portion of all the code in real world applications today is written in languages that are not memory safe like C or C++, for reasons that we will discuss in a later section. Because of this, they are vulnerable to exploits similar to those explored in this section.

For example, let's consider recent security advisory updates to Apple's macOS, Google's Chrome, and Mozilla's Firefox.

In Apple's most recent security advisory updates from October 31st, 2017 and December 6, 2017, there were a total of 26 memory related fixes out of 44 total security fixes in October and 14 out of 20 in December.

In Google Chrome's most recent security update for Chrome 63, of the fixes that were contributed by external researchers 9 of 19 mention some sort of memory corruption issue.

In the most recent Firefox security updates for Firefox 57, there have been 4 critical exploits related to specific attacks mentioned in this paper such as use-after-free, buffer overflows, and other memory corruption bugs that: "showed evidence of memory corruption and we presume that with enough effort that some of these could be exploited to run arbitrary code."

As we can see from the Firefox security updates, not only are memory related exploits still prevalent in modern software, but they are also highly critical and can allow for the worst case scenario of arbitrary code execution. In 2013, the internal emails of Hacking Team, a blackhat company that purchases zero-day exploits (ie. exploits that are unknown to the software vendor), were leaked to the public. In an analysis of these leaked emails, all the exploits that the Hacking Team ended up purchasing for tens of thousands of dollars were memory safety exploits like use-after-free and integer overflows.

Now that we have seen some concrete examples of memory related security exploits and understand that they really do cause the most severe security issues in real world code, we will now explore how to guard against such exploits.

4 DEFENSES AGAINST MEMORY EXPLOITS

In this section, we will begin by discussing some of the non language level defenses that can be added to a non memory-safe language. These consist platform level defenses and runtime level defenses that aim to defend against exploits in vulnerable code and static analysis to detect these vulnerabilities before even running the program.

Next, we will explore the current status quo of programming level defenses and why we still use memory unsafe languages when they are so prone to security vulnerabilities.

4.1 Platform Level

The unifying idea behind platform level defenses is to not allow for the exploit code to be run in the first place.

4.1.1 *DEP: Non-executable memory.*

Many of the exploits we covered in the previous section rely on being able to divert the control flow of a vulnerable program to a memory address whose contents an attacker controls in order to execute arbitrary code. Therefore, a sensible idea to guard against this is to set stack and heap memory as non-executable. In this case, even if an attacker was able to divert control flow to some malicious memory address, the computer would refuse to run the code from this memory address.

While this was a reasonable idea, unfortunately there are severe limitations such as requirements to have an executable heap in certain programs (ie. programs that rely on JIT compilation, for example JavaScript engines in web browsers). In addition, there is a technique called Returned Oriented Programming (ROP) that can severely undermine the efficacity of marking stack and heap memory as non-executable.

In a traditional memory related control flow hijacking attack, the attacker will somehow manage to store the malicious code he wants to execute somewhere in memory. In a ROP attack, the attacker will skip this step and instead focus only on diverting the control flow of the program to an existing set of instructions (ie. in a shared system library). These instructions (often called gadgets) are then chained together in order to create the full exploit code.

4.1.2 ASLR: Address Space Layout Randomization. In order to defend against ROP attacks and continue to strengthen the platform level defenses, the next step was to add Address Space Layout Randomization (ASLR). This defense technique randomizes stack addresses, heap addresses, and shared library addresses in order to increase security through obscurity. If the attacker can't reliably know where to divert the control flow to, then it will be harder to create a working exploit.

However, the downfall of ASLR is that almost any information leak will lead to being able to bypass ASLR. If we have a dangling pointer whose address we can know at runtime, then based on this information it may be possible to determine the randomized locations of everything else by computing it relative to the known address. This allows for exploits that are completely reliable and work every time. Another important flaw in ASLR is that this defense only works if it is applied consistently the executable and all of its related libraries. If a non-ASLR library is loaded, then this memory will be allocated deterministically and can be used to create a reliable exploit.

Nevertheless, even without an information leak to bypass ASLR, it is possible to write exploits that are not guaranteed to work. Of course, a brute force attack is theoretically possible. However, this is highly unlikely to actually work in practice as the space of memory addresses is much to large for this to work with any degree of reliability. We can improve this by using a "NOP slide" in order to increase the size of the target. A NOP slide is simply many no op instructions in a row so that we don't have to "land" directly on the start of the exploit code, but instead can land anywhere in the NOP slide and continue on to the exploit code. This will increase the probability of a brute force attack and can help, especially on 32 bit architectures which have lower total number of addresses. Another technique called partial overwrites relies on the fact that ASLR only randomizes the higher order bits of an address. In this way, if we can find a "good" instruction like `jmp` within the scope of this range of addresses, then it is possible to construct an exploit that will be variably reliable depending on how many higher order bits we need to use.

4.2 Runtime Level

The unifying idea behind runtime level defenses is to detect when an exploit is being run and stop execution.

4.2.1 Stack defenses. Stack canaries are a defense that target buffer overflows in particular. If we add a randomized canary to the end of the stack frame, in between the local variables and the saved frame pointer and return address, then it is possible to detect when this stack canary has been overwritten (ie. by a buffer overflow) and then simply crash the program. In more detail, at the start of the program's execution a random value is chosen for the stack canary which is then inserted into every stack frame. Before returning from every function, the stack canary is verified. Verifying the stack canary at this time makes sense as many buffer overflow attacks attempt to overwrite the saved return address in the current stack frame.

Another idea in order to make stack based overflows more difficult is to rearrange the layout of the stack by having the local variables section of the stack grow downward away from the arguments, saved return address and saved stack frame. In addition, a stack canary is still used in between these two sections.

Although these defenses seem like they would work quite well, they simply increase the difficulty of creating exploits. In particular, one example of bypassing stack canaries is to take advantage of how Windows handles exception handling. In the stack frame, exception handlers are placed above the canary and before the saved stack frame pointer and return address. If we can control an exception handler, then when an exception is triggered we can jump directly into our exploit code without checking the stack canary!

In addition, it can also be possible to extract the value of the canary if it remains unchanged. For example, we can consider a server who restarts a process automatically upon crashes using `fork()`. This means that the canary will continue to have the same value and we can extract the random value by continually trying to modify bytes of the canary one by one until we do not crash the program. If the program does not crash, then we have successfully extracted that byte of the canary and we can move onto the next byte until we have extracted the entire canary.

One last stack defense called StackShield works by keeping a copy of the saved return address and saved stack frame pointer in a safe place in memory that is difficult to corrupt such as in the stack's data segment. During the function return process, check to make sure that the current saved return address and saved stack frame pointer match the saved copies. However, overwriting the saved return address is not the only way to hijack the control flow of a program. For example, we can modify the Global Offset Table which deals with the position of system calls in memory. If a system call is used in the program, we will modify its address in the Global Offset Table to our own shellcode and then we can hijack control without modifying either the saved return address or the saved stack frame pointer.

4.2.2 Control Flow Integrity. A more general approach to preventing control flow hijacking attacks is Control Flow Integrity. This defense technique uses static analysis of a program's source code in order to construct a control flow graph that represents valid control flows for the program. In theory, this can prevent malicious control flow hijacking as they would not appear on the control flow graph and therefore be blocked. While this seems great in theory, in practice existing control flow integrity are not very practical because the static analysis portion requires intimate knowledge of the inner workings of the software, which is not always available for commercial products. In addition, it can impose a large performance overhead which can be anywhere from 25% to 50%. Finally, as with the other techniques discuesd in this section, CFI is not a perfect defense and can still be bypassed using a more advanced version of

Return Oriented Programming that uses gadgets that the control flow graph considers valid.

4.3 Static Analysis

The last defense I will briefly cover before going into language level defenses is static analysis tools such as Coverity. These tools, similar to Control Flow Integrity, aim to statically analyze the code for security vulnerabilities and other bugs. Their website specifically names some memory safety related issues such as "dereference of NULL pointers", "buffer overflows", and "use of resources that have been freed".

Of course, such tools will not be able to fully understand the complexity of every program, but they are able to successfully find more obvious issues. Another flaw is that they rely on identifying known exploits, so vulnerabilities that are not yet well known to the security community may not be found in static analysis.

4.4 Language Level

Although platform level defenses, runtime level defenses, and static analysis increase the difficulty of exploiting a lack of memory safety, as we saw when discussing these defenses, they simply create a constant state of war between creating a new defense and attackers come up with a more advanced way to exploit them. In addition, almost all of these aforementioned defenses incur some performance cost which can make them impractical.

Instead of relying on these somewhat ad-hoc solutions that attempt to add memory safety to programs written in memory unsafe languages, it seems logical to add memory safety guarantees at a more fundamental level. If we only used memory safe programming languages, then many if not all of these exploits would not even be possible as the programming language itself would prevent us from writing programs with these memory safety related vulnerabilities.

The question we need to ask ourselves now is why is there so much memory unsafe code still out there? The reason is that programming languages lie on a spectrum of safety vs. control. Most modern languages today are memory safe because they rely on automatic memory management techniques like garbage collection. Although this does not necessarily mean reduced performance as modern GC techniques such as generational GC can be just as fast as manual memory management, because the process is automated these GC'd languages cannot offer the same amount of raw control as manually memory managed languages like C or C++. In addition, although performance may not be an issue, many garbage collected languages also impose a runtime requirement which increases the amount of computing resources needed. These resources may not be available on certain devices such as embedded systems. Finally, some applications such as operating systems really do need to control every aspect of memory management. In a GC'd world, there is some element of randomness because the memory management is automatic. Deterministic memory management can only be achieved with languages that offer full control.

Despite the fact that memory unsafe languages are still very prevalent today which leads to many security issues, we have still come a long way from the days of pure C. In particular, C++ offers many memory related abstractions through a system of smart pointers and move semantics which allow for a much "safer" programming experience. However, C++ is still a fundamentally memory unsafe programming language because although these safe coding standards exist, there is no enforcement of these rules at the language level. In addition, these coding standards do not prevent all types of memory safety issues. In particular, dangling references, use after free, and iterator invalidation are still quite common, even in C++ code that adheres to these modern standards.

In conclusion, although language level defenses have come a long way since the age of writing everything in C, there will always be a need for a programming language that offers full control over memory management. While modern C++ has abstractions that allow for more safety than plain old C, these coding standards are not enforced by the language and do not prevent all types of memory safety issues. In the next section, we will explore how Rust is unique because it offers both guaranteed memory safety at the language level and full control.

5 RUST

In the introductory slides to Rust, the creator of Rust, Graydon Hoare, calls Rust "Technology from the past, come to save us from the future" because Rust leverages programming language techniques that have already existed before in a unique way to resolve issues such as memory safety and concurrency that are the bane of systems programmers. From its conception, Rust was designed to be a memory safe systems programming language by combining the memory safety of modern, GC'd languages with the absolute control and zero-cost abstractions of systems programming languages like C or C++.

5.1 How does Rust enforce memory safety?

Let's begin our discussion of Rust by seeing how memory safety is enforced at a language level. Unlike C++ where memory safety abstractions exist but are not enforced by the language, in Rust memory safety is guaranteed by the type system and rigorous bound checking.

As seen in lecture, the concepts of ownership, borrowing and lifetimes allow Rust to avoid aliasing (multiple references to same object) and mutation occurring at the same time. To briefly recap:

- Ownership (type `T`): every resource has a unique owner who cannot free or otherwise mutate its resource while it is borrowed.
- Shared borrow (type `&T`): multiple entities can borrow an object from the owner at the same time, but they cannot mutate the object.
- Mutable borrow (type `mut &T`): a single entity can borrow an object from the owner with mutation privileges, thus avoiding aliasing.

Finally, the concept of lifetimes allows us to reason temporally about ownership and borrowing. For example, the lifetime of an object tells us when a borrowed object will be returned to its owner or an object will no longer be valid. In Rust, the lifetime of a value is tied to the lexical lifetime of its name.

Although the idea of avoiding simultaneous mutation and aliasing is a good way of explaining memory safety in Rust, it does not map exactly to our previous definition of avoiding all spatial and temporal memory errors that we used in previous sections.

In terms of avoiding spatial errors, Rust's type system allows for a great deal of compile-time and runtime bounds checking. For example, the array type that is often exploited in C stores both type information and length. If we try to access or write to an index that is out of bounds, Rust will panic at runtime and crash the program, resulting in a denial of service instead of a security vulnerability. Although bounds checking does incur some runtime cost, using iterators instead of a for loop will avoid these runtime costs.

If we recall the 1-byte overflow example, we can construct a similar example in Rust as follows:

```
fn main() {
  let mut buf = [1, 2, 3, 4];
  for i in 0..buf.len()+1 {
    buf[i] = 0x90;
  }
}
```

Running this code will result in the following error message:

```
thread 'main' panicked at 'index out of bounds:
the len is 4 but the index is 4', tmp.rs:9:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Similar bounds checking exists for heap allocated values therefore also avoiding spatial errors related to heap overflows.

In terms of avoiding temporal errors, the ownership and lifetimes in Rust will not allow for these types of errors to occur. For example, if we consider the insidious use-after-free vulnerability, Rust will catch this error at compile-time by determining that the object does not live long enough to be used at that point:

```
fn take<T>(_x: T) {}

fn main() {
  let x = Box::new(0x90);
  take(x);
  println!("{}", x);
}
```

In this example, the `take()` function will take ownership of `x` and therefore free `x` upon the end of the function. This will result in the following compiler error:

```
error[E0382]: use of moved value:  x
[...]
```

On the other hand, a similar function in C:

```
int take(int *x) {
  free(x);
  return 0;
}

int main() {
  int *x = (int *)malloc(sizeof(int));
  *x = 42;
  take(x);
  printf("%d\n", *x);
  return 0;
}
```

This will compile just fine and in fact print 42 as output.

5.2 Proving Rust's safety guarantees

In order to conclude this section on the guaranteed memory safety of Rust, we can begin by questioning whether or not an ownership-based type system is enough to guarantee memory safety. Although it seems to work in practice, is it possible to mathematically prove the safety of Rust? Another question one might have at this point is that we have not yet discussed the elephant in the memory safety room: `unsafe`.

Even if we assume that Rust's type system is enough to guarantee memory safety, the `unsafe` keyword allows the programmer to bypass these type safety checks. Essentially, the programmer is telling the Rust compiler to trust that the code enclosed in the `unsafe` keyword is really safe even though the type system might not be able to understand it. This is also necessary in order to interface with other languages like C through a FFI.

Although it may seem surprising that a language with such a literally unsafe feature can still be guaranteed to be memory safe, it is in fact possible to prove Rust's safety guarantees even with unsafe, with the caveat that unsafe code blocks must be fully encapsulated. This means that if someone else uses these functions containing `unsafe` but do not use `unsafe` themselves, then their code will be free of any unsafe or undefined behaviors. This is particularly important as much of Rust's core infrastructure uses `unsafe` in its implementation, so if it were not possible to guarantee safety if `unsafe` is ever used, then the entire memory safety guarantee would crumble.

In the paper "RustBelt: Securing the Foundations of the Rust Programming Language", the authors provide an extensible proof of Rust's soundness by giving a formal and machine-checked proof for the soundness of a realistic subset of the Rust and a few of its most important libraries. This proof is extensible because it provides formal guidelines for the requirements of any new `unsafe` libraries that are added to the Rust language. This formal proof actually led to fixing a real soundness issue in the Rust language and provides a reasonable degree of proof that the Rust language itself is sound.

6 CONCLUSION

In conclusion, we can now see why memory safety is important because of its severe implications for security. The lack of memory safety is the leading cause for control flow hijacking attacks that are still very prevalent even in today's modern software. Although defenses exist at a platform or runtime level, these ad-hoc defenses that are trying to fix an inherently unsafe programming model are not sufficient to stop all attacks and there will always be a constant war between attacker and defender. In addition, these defenses come at a performance cost and some are too costly to even implement practically.

A better solution would be to continue to adopt memory safe languages at all levels, including systems levels languages which have notoriously been slow to change from their origin as nicer assembly language. Unlike the ad-hoc defenses at a platform or runtime level, language level memory safety tackles the root cause of memory safety issues by not allowing memory unsafe programs to be written in the first place. Despite the fact that most modern languages are memory safe, there is and will always be a necessity to have a systems programming language that offers full control over resource management. Until recently, there has been no mainstream language that offers both memory safety and full control so C and C++ have continued to dominate this space. However, I believe that languages like Rust that provide both safety and control will and should continue to grow in popularity as we realize that it is almost impossible to maintain any guarantee of safety and therefore security in existing systems programming languages.

7 REFERENCES

- Boiling the Ocean, Incrementally - How Stylo Brought Rust and Servo to Firefox
  http://bholley.net/blog/2017/stylo.html
- What does Rust's "unsafe" mean?
  http://huonw.github.io/blog/2014/07/what-does-rusts-unsafe-mean/
- SoK: Eternal War in Memory
  http://ieeexplore.ieee.org/document/6547101/?arnumber=6547101
- Out of Control: Overcoming Control-Flow Integrity
  http://ieeexplore.ieee.org/document/6956588/?section=abstract
- Smashing The Stack For Fun And Profit
  http://phrack.org/issues/49/14.html
- The Frame Pointer Overwrite
  http://phrack.org/issues/55/8.html
- Once upon a free()...
  http://phrack.org/issues/57/9.html
- Advanced Doug lea's malloc exploits
  http://phrack.org/issues/61/6.html
- MALLOC DES-MALEFICARUM
  http://phrack.org/issues/66/10.html
- Modern Binary Exploitation - CSCI 496
  http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/15/09_lecture.pdf
- Let's sunset C/C++
  http://trevorjim.com/lets-sunset-c-c++/
- Off-by-one Overflow
  http://users.own-hero.net/~decoder/offbyone.pdf
- Project Servo: Technology from the past, come to save the future from itself
  http://venge.net/graydon/talks/intro-talk-2.pdf
- Bypassing Browser Memory Protections
  http://www.hakim.ws/BHUSA08/speakers/Sotirov_Dowd_Bypassing_Memory_Protections/BH_US_08_Sotirov_Dowd_Bypassing_Memory_Protections.pdf
- Vudo - An object superstitiously believed to embody magical powers
  http://www.phrack.org/issues/57/8.html#article
- CS201 Computer Systems Programming
  http://www.thefengs.com/wuchang/courses/cs201/
- Guaranteeing memory safety in Rust
  https://air.mozilla.org/guaranteeing-memory-safety-in-rust/
- Stable Channel Update for Desktop
  https://chromereleases.googleblog.com/2017/12/stable-channel-update-for-html
- CS155 Computer and Network Security
  https://crypto.stanford.edu/cs155/syllabus.html
- The Rust Programming Language
  https://doc.rust-lang.org/book/second-edition/
- Rust is mostly safety
  https://graydon2.dreamwidth.org/247406.html
- Preventing Use-after-free with Dangling Pointers Nullification
  https://lifeasageek.github.io/papers/lee:dangnull.pdf
- Modern C++ is not memory safe
  https://news.ycombinator.com/item?id=7587978
- RustBelt: Securing the Foundations of the Rust Programming Language
  https://plv.mpi-sws.org/rustbelt/popl18/
- Coverity Scan FAQ
  https://scan.coverity.com/faq
- Apple security updates
  https://support.apple.com/en-us/HT201222
  https://support.apple.com/en-us/HT208221
  https://support.apple.com/en-us/HT208331
- It's time for a memory safety intervention
  https://tonyarcieri.com/it-s-time-for-a-memory-safety-intervention
- Memory Errors: The Past, the Present, and the Future
  https://www.isg.rhul.ac.uk/sullivan/pubs/tr/technicalreport-ir-cs-73.pdf
- Security Advisories for Firefox
  https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/