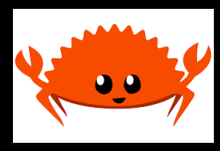
The Beauty of Rust

Safe, Secure, Systems Programming

Peter Zdankin

Introduction

Who me?

- I’m Peter Zdankin, PhD Student at Verteilte Systeme and researching IoT Longevity
- Teaching experience since 2014, Rust experience since 2016
- Here to preach about the holy crab Ferris  (Mascot of Rust)
- Feel free to interrupt me and ask questions, I like to explain stuff!

Introduction

Who you?

- Computer Science/ Komedia/ ISE Students?
- Bachelor/Master?
- Do you know other systems programming languages (C/C++/..) ?
- Are you able to write programs in at least one other language?
- What do you expect of this course?
- What made you visit this course?

Organisation

- Completely voluntary course, I don’t get paid, you don’t get credit points
- Wednesdays 16-18 here (BC 012)
- Course Material: https://github.com/TheRustyStorm/BeautyOfRust
- Questions: peter.zdankin@uni-due.de

Any other questions?

A programming language allows to translate the programmers intent into machine-interpretable commands

Programming Languages

- Translating intents into machine language is hard..
- Different levels of abstraction allow for more/less control over the way this is achieved
- Higher abstracted languages (Python, C#, Java, Go etc.) leave less control, and therefore less room for errors
    - The devil is in the details
- Less control -> Fewer Errors ^ Less Optimal solutions

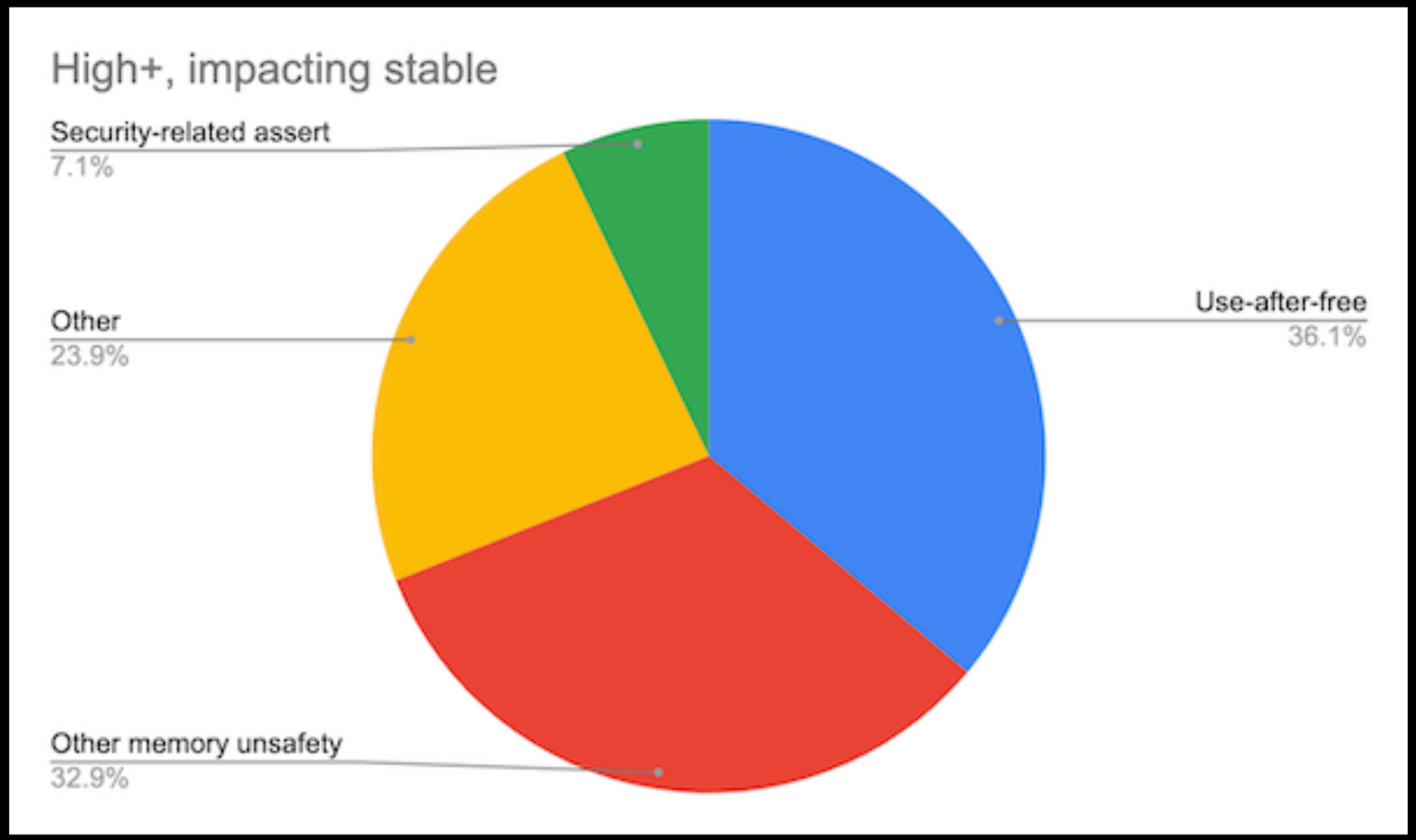
Cont’d

- If we want more optimal solutions, we need to have more control over the translation process
- With more control, there is more room for errors
  - Incorrect buffer access
  - Dynamic memory management errors
  - Incorrect assumptions about the target hardware
- „But I’m a good programmer, only idiots make these mistakes.. obviously!“

100%

of programmers think so

Chrome


High+, impacting stable
Security-related assert
7.1%
Other
23.9%
Use-after-free
36.1%
Other memory unsafety
32.9%

Fish in a Barrel
@LazyFishBarrel

Thanks to Google's detailed technical data we can provide total memory unsafety statistics for public 0days by year:

2014 5/11 45%
2015 22/28 79%
2016 22/25 88%
2017 17/22 77%
2018 12/12 100%
2019 9/10 90%

Total 87/108 81%

#memoryunsafety

Buffer Overflow enabled SQL Slammer

Flashback Friday: SQL Slammer

Within a few hours of being released in the winter of 2003, SQL Slammer had brought the internet to something of a standstill. We look back at this notable worm.

Trident-Schwachstellen: Alle Technischen Details An Einem Ort

November 2, 2016

Lookout

Heute **veröffentlichtLookout die technischen Details hinter "Trident",** einer Reihe von iOS-Schwachstellen, die es einem Angreifer ermöglichen, das Gerät eines Zielnutzers aus der Ferne zu knacken und Spyware zu installieren. Im August entdeckte Lookout in Zusammenarbeit mit Citizen Lab "Pegasus", eine ausgeklügelte mobile Spyware, die von staatlichen Akteuren zur Überwachung hochrangiger Ziele eingesetzt wird. Der so genannte "Cyber-Waffenhändler" NSO Group hat die Spyware entwickelt, die sich

MARCH 17, 2015

Would Rust have prevented Heartbleed? Another look

In case you haven't heard, another serious OpenSSL vulnerability will be announced this Thursday. It reminded me of about a year ago, when Heartbleed was announced:

Wednesday, September 16, 2015

Stagefrightened?

Posted by Mark Brand, Bypasser of Mitigations

There's been a lot of attention recently around a number of vulnerabilities in Android's libstagefright. There's been a lot of confusion about the remote exploitability of the issues, especially on modern devices. In this blog post we will demonstrate an exploit for one of the libstagefright vulnerabilities that works on recent Android versions (Android 5.0+ on Nexus 5).

The vulnerability (CVE-2015-3864) that we've chosen to exploit is an imperfect patch for one of the issues reported by Joshua Drake, which has been fixed for Nexus devices in the September bulletin. Several parties noticed the problem, including at least Exodus Intel and Natalie Silvanovich of Project Zero. It's a promising looking bug from an exploitation perspective: a linear heap-overflow giving the attacker control over the size of the allocation; the amount of overflow, and the contents of the overflowed memory region.

The vulnerable code is in handling the 'tx3g' chunk type when parsing MPEG4 video files. Here's the original vulnerable code:

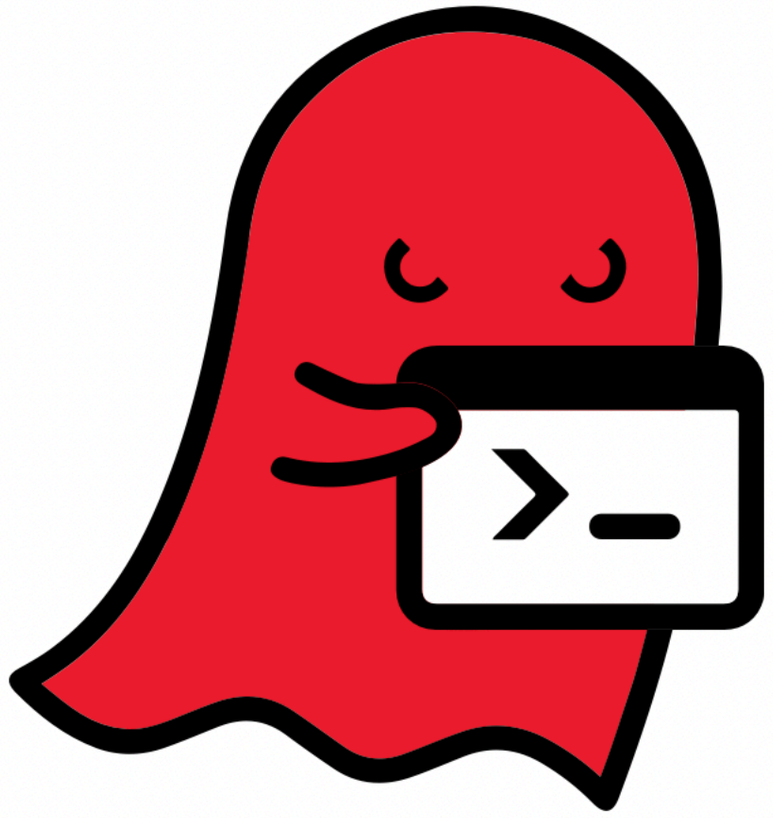
Note when reading that `chunk_size` is a `uint64_t` that is parsed from the file; it's completely controlled by the attacker and is not validated with regards to the remaining data available in the file.

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
    uint8_t *buffer = new uint8_t[size + chunk_size]; // <---- Integer overflow here
    if (size > 0) {
        memcpy(buffer, data, size);                   // <---- Oh dear.
    }
    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
```

The GHOST Vulnerability

Amol Sarwate
January 27, 2015 - 4 min read

The GHOST vulnerability is a serious weakness in the Linux glibc library. It allows attackers to remotely take complete control of the victim system without having any prior knowledge of system credentials. CVE-2015-0235 has been assigned to this issue.



Qualys security researchers discovered this bug and worked closely with Linux distribution vendors. And as a result of that we are releasing this advisory today as a co-ordinated effort, and patches for all distribution are available January 27, 2015.

QualPwn vulnerabilities in Qualcomm chips let hackers compromise Android devices

Patches for the QualPwn vulnerabilities have been released earlier today by both Qualcomm and the Android team.

Memory Unsafety in Apple's Operating Systems

JULY 23RD, 2019

Over at @LazyFishBarrel we've been tweeting statistics about the proportion of memory unsafety related vulnerabilities in various software for a little over a year[1]. However, while Twitter is a great medium for dispensing this information on a per release basis, it's not great for deeper analysis. Rather than just talking about a single release, what if we aggregated the total memory unsafety-related vulnerability statistics in Apple's two flagship operating systems: iOS and macOS?[2]

What is memory unsafety and why should I care?

Memory unsafety[3] is a property of a programming language that allows the creation of bugs and security vulnerabilities related to memory access. Languages like C and C++[4] are memory unsafe because they allow memory violations such as use of uninitialized memory, double free, buffer overflow, use after free, etc. To avoid these the programmer must perfectly allocate, write, read, and deallocate memory or else serious vulnerabilities can easily occur. These bugs are especially frustrating because they represent an class of issues that can be **entirely fixed** by moving to languages that do not suffer from these limitations[5].

iOS 12

iOS 12 was released September 17, 2018 and has subsequently had a total of 11 feature and point releases. Unlike macOS, the iOS 12 series has several small bugfix releases that did not correct security issues. These are marked as N/A for bugs/CVEs but remain present in the list below.

Total CVE Count	Memory Unsafety Bugs	Percentage	Release
37	28	75.7%	12.4
N/A	N/A	N/A	12.3.2
N/A	N/A	N/A	12.3.1
42	34	81%	12.3
51	29	56.9%	12.2

Only big companies?

- Survivorship bias, you only hear about these kinds of bugs if they are big enough to get a headline
- Nobody cares about security issues with your grandmas t-shirt shop app
- Chances are most code written in C is buggy as hell

„You save time when you don't need to have an awards ceremony every time a C statement does what it's supposed to do. Moreover, higher-level languages are more expressive than lower-level languages.“

http://www.amazon.com/exec/obidos/ASIN/0735619670/codihorr-20

What are some key causes of errors?

- Race conditions
- Memory leaks/invalid access
- Buffer overreads/overwrites
- Multithreading issues

Let’s inspect some of them

Race Conditions

- Occur if more than one piece of code wants to access shared data
- Multithreading, or program flow

```
std::vector<int> v = {10,20,30,40,50};
for(int i: v){
        for(int x = 0; x < i; x++){
            v.push_back(x);
        }
    std::cout << i << ' ';
}
```

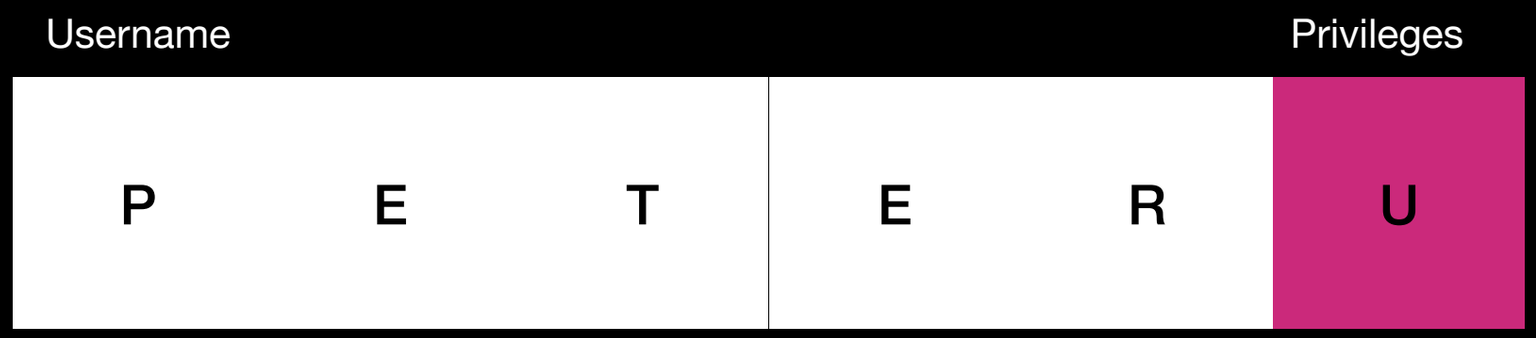
- As soon as several places have mutable access, it’s quite dangerous

Memory Leaks/ Invalid Access

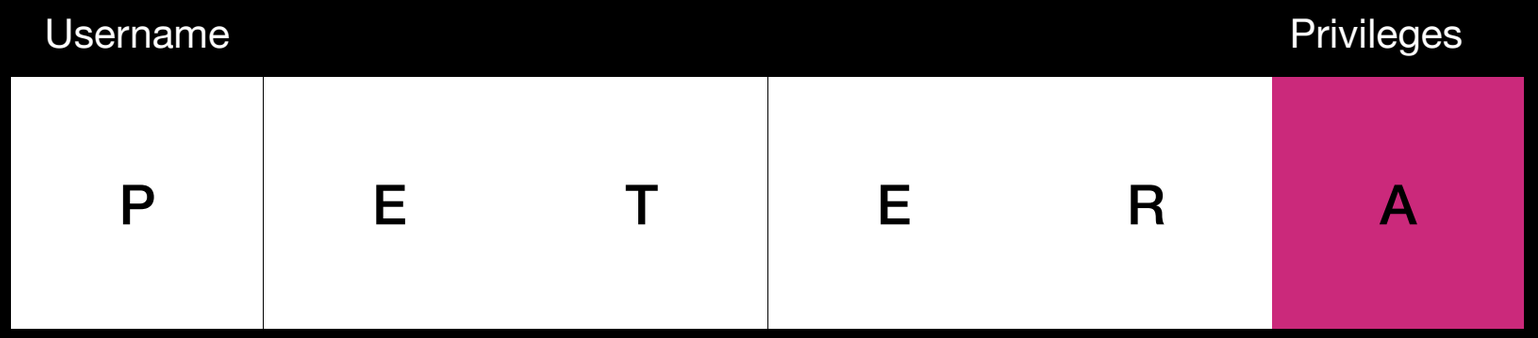
- A memory leak happens when a memory region is allocated but never returned
- Garbage collectors in highly abstracted languages help with this
- In lower abstraction languages you must guarantee that every program flow ensures the freeing of allocated resources
- Invalid access happens if you access resources that are not yet allocated correctly
  - NullPointer
  - Segmentation Faults

Buffer Overreads/Overwrites

- A specification of invalid access is an overread/overwrite
- If you have a fixed-sized memory region and access out of its bounds, weird things can happen and will be exploited
- Example: Store username and privilege in memory



- Enter username that is too long and writes over privileges field Name: PETERA (A for admin, U for user)
-



Multithreading Issues

- Most industry used programming languages were created before there was multiprocessing
- Threading was supported through a language update
- Backwards compatibility had to be maintained
- Usually locking/mutex and other constructs are used in a multithreaded context
- But most often you have to remember to lock/unlock data, thus errors can happen

What if we go back to the drawing board, and make things right this time?

Rust

- Created by Mozilla as a safe systems programming language
- Ownership model, you can only borrow or move ownership
- Composition instead of Inheritance
- Strongly typed
- Strict Compiler and many rules by the language
- Feels restrictive at first, but once you get into it you’re on high speed

You must program

You must program

- Throughout the course we will have a lecture and an exercise slot.
- I guarantee you personally, that if you do not program in Rust, you will never learn it.
- We will do live-coding, I will go around and explain stuff and help with issues
- You must literally and physically have a laptop in front of you and attempt to solve the exercises, or else this is a waste of time

The last steps for today

1. Open my Github https://github.com/TheRustyStorm/BeautyOfRust
2. Install Rust through rustup
3. Install Text Editor (I recommend VS Code)
4. cargo new hello
5. src/main.rs

```rust
fn main() {
    println!("Hello, Peter!");
}
```

6. cargo run