Embedded System Security with Rust

Case Study of Heartbleed

Jens Getreu, Tallinn University of Technology,

Revision History

Revision 1.1	17.2.2016	JG

Table of Contents

Abstract: Real-time embedded systems have to meet a combination of requirements that are in deep tension: they are expected to deliver timely results, observing strict deadlines, all using only very limited resources, computing power and energy. To this day, the most common programming language in this environment is *C/C++* because of its zero-cost abstractions and fine control over memory layout. With the upcoming communication ability through network interfaces, an additional requirement gained in importance: security. Unfortunately, *C/C++* supports some secure software design principles only rudimentary. Too many very severe vulnerabilities are directly related to the fact that *C/C++* does not guarantee memory safety. To exemplify this, the prominent Heartbleed vulnerability is discussed in terms of causes, technical details and impact.

A new programming language *Rust*, originally designed to develop the successor of the Firefox web browser, comes with a couple of innovative features. The author argues that *Rust*, inter alia for its memory safety, is well suited to succeed *C/C++* in embedded system programming. This is demonstrated by reproducing the Heartbleed vulnerability in *Rust* and by observing how the system responds to this kind of attacks.

Keywords: Embedded System Security, Heartbleed Bug, Rust Language

1. Introduction

Although embedded systems have existed since the early age of computing, system security was not an issue in the beginning. Originally, embedded systems had no network connection and no interface to the outside world beyond their system operators. Of course, also in those times, programs had flaws but some errors never did show up in the production cycle. For some errors,the operators found work-arounds and only very few actually had to be fixed by the manufacturer after device delivery.

The situation dramatically changed with the upcoming of network interfaces on embedded devices. Before, only well-meaning operators could interact with the software. Now, network interfaces expose parts of the API to everyone who is able to send a network package to the target system. As all systems are interconnected these days it is possible to attack any embedded device even through well protected networks. A very impressive example is the *Stuxnet* exploit discovered in 2010, used to destroy up to one-fifth of Iranian centrifuges and to delay that country's nuclear ambitions. One of the main attack vectors was a vulnerability in Windows called CVE-2010-2568. It took Microsoft not less than 4 years to release a patch *MS15-020* which is believed to finally address the issue [1].

In pre-network-times, the impact of software flaws in embedded systems was usually limited to impair some secondary functions. For example, I once had a car with a software flaw that switched on the ceiling light under some non-reproducible coinciding events. For the driver this flaw was annoying but without any further consequences. Or, let's consider a software flaw in a television set that impairs a secondary function under certain conditions. The same television set once connected to the Internet might be hacked using this vulnerability. The attacker then installs some listening software in your living room or might use the TV as point of departure for further attacks. Overall, the upcoming of the *Internet of Things* will put more and more interconnected computers into all sorts of consumer devices - and our living rooms.

2. Embedded Devices Are Vulnerable

Why does the presence of a network interface on an embedded device make such a difference? Because it potentially connects people with malicious

intent to our device. A well-meaning operator tries to work-around software flaws while attackers systematically scan our systems to find and exploit vulnerabilities. Consequently, when an Internet connection is supported, security is of utmost concern.

Schneier [2] compares this situation with what we have seen in the 1990s when the insecurity of personal computers was reaching crisis levels: “Software and operating systems were riddled with security vulnerabilities, and there was no good way to patch them. Companies were trying to keep vulnerabilities secret, and not releasing security updates quickly. And when updates were released, it was hard — if not impossible — to get users to install them”. How well this applies to today’s embedded devices is best shown with home routers. These embedded systems are more powerful then the PCs of the 1990s and have became as such a popular target for attackers. A recent case concerned 1,3 Mio. Vodaphone clients in Germany [3]: A so called *WPS Pixie Dust* vulnerability [4] found in a Wi-Fi setup function allows attackers to hack the routers WPA password.

The security researchers Runa Sandvik and Michael Auger managed to hack a sniper rifle via it’s Wifi-interface [5]: The TP750 is a computer assisted self aiming long range sniper rifle. Its targeting system guarantees almost foolproof accuracy by firing not when the shooter first pulls the trigger but instead only when the barrel is perfectly lined up with the target. A chain of vulnerabilities allowed the attacker to take control over the self-aiming parameters and to deviate the bullet to any arbitrary sufficiently close target.

3. Causes

Schneier [2] argues that embedded devices are insecure mainly because the software is often unpatched and much older than the last maintained branches of the software deployed. It is important to note that there are many entities involved in a typical embedded system design, manufacturing, and usage chain. Often neither the chip nor the device manufacturer is motivated to maintain his firmware and to publish updates. As there is generally no patch distribution infrastructure in place, the user has to install the patch manually. Even if a patch is available it is hard to get users to install it, partly because they are is not aware of patch’s availability, and partly because the firmware update procedure is complicated.

Will the situation change? From the manufacturer's point of view security is a costly service. Finding the right balance between cost and benefit of security is not always easy. But more and more users have become aware of the risks. Also most of the big market leaders have experienced that hacked end user devices cause a considerable damage to the brand's reputation. Such a loss of reputation is then very costly to recover. But the situation may change soon: Big telecommunication companies like Deutsche-Telecom and others already improved their patch policy and distribute vulnerability reports and patches for their routers. Let's hope that the manufacturing of other embedded devices follow soon.

Manufacturers usually complain that their profit margin is too small and consumers are not ready to pay for security. This might be true in many cases but in accordance with fundamental marketing principles, complex services have to be explained to the customer! So far no one really does. Also a non-tech-savvy user is able to understand that error free software does not exist. So instead of hiding software flaws, manufacturers should advertise their bug fixing infrastructure and service with confidence.

Complex software without regular security-bug-fixes is in a similar bad state as political systems without regular disclosure of corruption cases.

4. Embedded System Development

Kopetz [6, p. 18] lists a number of distinctive characteristics that influences the embedded real-time system development process:

- Mass production market
  The cost per unit must be as low as possible and efficient memory and processor utilization is of concern. In order to reduce costs, embedded systems are highly specialized i.e. designed to meet a limited well defined set of requirements.
- Static structure
  The known priory environment is analysed at design time and considered to be static. This allows us to simplify the software, increases the robustness and improves the efficiency of the embedded device.
- Maintenance strategy
  The hard- and software of many systems is not designed for maintenance.
- Ability to communicate

More and more intelligent products are required to communicate over networks with some larger system.

- Limited amount of energy
  Many devices are powered by batteries.

All the above design principles are concurrent with security requirements. For example, in order to minimize production costs, embedded systems are highly specialized machines: For a limited set of inputs they produce a well defined output. In contrast, secure machines are *generalistic*. They must deal securely with all possible network inputs. Another important requirement is low energy consumption, often limiting the computing power and thus excluding the deployment of security software like intrusion detection or malware scanner on most embedded systems.

The art of system design consists in finding well-balanced priorities in meeting different partly concurrent requirements: Figure 1, “Dependability and security attributes” shows the meta-functional attributes of a computer system related to *dependability* as quality of service and to security comprising *confidentiality*, *integrity* and *availability* [7, Fig. 2.1]. Most designs prioritize the attribute *availability* because it is the most obvious to the client. It does not mean that the other attributes are fully ignored, but in general much less production resources are deployed to meet them.

Figure 1. Dependability and security attributes

- Legend Figure 1, “Dependability and security attributes”: *Availability*: readiness for correct service. *Reliability*: continuity of correct service. *Safety*: absence of catastrophic consequences on the user and the environment. *Integrity*: absence of improper system alterations. *Maintainability*: ability to undergo modifications, and repairs. *Confidentiality*: i.e., the absence of unauthorized disclosure of information.

The most neglected attribute of embedded systems is *confidentiality*. Historically non-significant, it gained importance with communication abilities through network interfaces. Nowadays, a loss of sensitive data can have disastrous consequences as shown in Section 5, “The Heartbleed Vulnerability”.

What can be done to make our embedded systems more secure? First, the system should be designed at any stage to deal with malicious input. To integrate this idea systematically in the development process, many security extensions to system modelling languages have been suggested: *BPNM*, *Secure Tropos*, *Misuse Cases*, *Mal-activity diagrams*, *UMLsec*, *SecureUML* and *Trust Trade-off Analysis* are some examples.

On the programming side many modern languages provide desirable security features like guaranteed *memory safety* or *data race freedom*. However, none of these was yet able to impose higher margins on the embedded systems market. According to a recent survey among developers about the languages used for the current project [8], 53% still use *C++* followed by 52% using *C*. The researchers expect that “the growth of *C++* to remain strong in the coming years, as object-oriented languages gain further acceptance within the development of safety-critical systems.” C was designed in the 1970s by Dennis Ritchie at AT&T Bell Labs. Though never intended for embedded use, *C* remains the most widely used embedded programming language. *C++* is an extension to the *C* language providing inter alia object oriented features. Because of their common language elements both languages share the same drawbacks. Neither *C++* nor *C* provide guaranteed memory safety: Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program’s behavior or take full control over it by hijacking its control flow. Even though the problem is well studied and partial remediations have been proposed, none found a broader acceptance [9, p. 48].

Considering the still widespread use of *C* and *C++* ,especially in embedded systems, it is not surprising that memory safety related weaknesses still rank in top positions in vulnerability statistics. The classic “buffer overflow” for example reached position 3 in the *CWE/SANS Top 25* of 2011. Table 1, “Common weaknesses in C/C++ that affect memory” shows the most common *C/C++* memory safety bugs.

Table 1. Common weaknesses in C/C++ that affect memory

CWE ID	Name
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
125	Out-of-bounds Read
126	Buffer Over-read ('Heartbleed bug')
122	Heap-based Buffer Overflow
129	Improper Validation of Array Index
401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')
415	Double Free
416	Use After Free
591	Sensitive Data Storage in Improperly Locked Memory
763	Release of Invalid Pointer or Reference

All memory-related problems in *C* and *C++* come from the fact that *C* programs can unrestrainedly manipulate pointer to variables and objects outside of their memory location and their lifetime. This is why memory safe languages like *Java* do not give programmers direct and uncontrolled access to pointers. The *Java* compiler achieves this with a resource costly runtime and a garbage collector. The additional costs in terms of resources eliminate this solution for most real-time embedded applications.

For many years program efficiency and memory safety seemed to be an insurmountable discrepancy. Now, after 10 years of development, a new programming language called *Rust* promises to cope with this balancing act. *Rust*'s main innovation is the introduction of semantics defining *data ownership*. This new programming paradigm allows the compiler to guarantee memory safety at *compile-time*. Thus no resource costly runtime is needed for that purpose. In *Rust* most of the weaknesses listed in Table 1, “Common weaknesses in C/C++ that affect memory” are already detected at compile time. Moreover *Rust*'s memory safety guarantees that none of

these weaknesses can result in an undefined system state or provoke data leakage.

To illustrate the matter we will analyse a typical *C/C++* memory safety related bug in Section 5, “The Heartbleed Vulnerability”. Then we reproduce the erroneous code in *Rust* and observe the system’s response in Section 6, “Could Heartbleed Have Happened With Rust?”

5. The Heartbleed Vulnerability

The Heartbleed vulnerability is an excellent example of a typical memory safety related weakness of the *C/C++* language: A “CWE-126: Buffer Overread” (cf. Table 1, “Common weaknesses in C/C++ that affect memory”).

5.1. The Bug of the Century

It sounds like science fiction: 4 missing lines in a computer program compromise at least one quarter of the Internet’s cryptographic infrastructure! The proportion of vulnerable Alexa Top 1 Million HTTPS-enabled websites have been estimated as lying between 24–55% at the time of the so called Heartbleed vulnerability disclosure [10, p. 4]. Another study found “that the heartbeat extension was enabled on 17.5% of SSL sites, accounting for around half a million certificates issued by trusted certificate authorities. These certificates are consequently vulnerable to being spoofed (through private key disclosure), allowing an attacker to impersonate the affected websites without raising any browser warnings [11].”

The Heartbleed vulnerability, listed as `CVE-2014-0160`, was publicly disclosed on 04/07/2014 with the following statement [12]:

> The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).
>
> The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used

> to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

The bug had been introduced with a minor SSL/TLS protocol extension called *Heartbeat* in January 2012. From the beginning the vulnerability had been widely exploited: For example, the U.S. National Security Agency NSA knew for at least two years about the Heartbleed bug, and regularly used it to gather critical intelligence [13]. As another example, a group of Chinese hackers referred to as “APT18” used Heartbleed to bypass the security systems of the US “Community Health Systems” company and stole 4.5 million patient records [14]. Estimating how widely Heartbleed was and is being exploited is difficult as the attack leaves no trace. The stolen keys are usually stored for later usage in more complex attack scenarios, as *Raxis*, an independent penetration testing firm, has demonstrated [15].

Even though the Heartbleed patch itself has only a handful of characters, its recovery is tremendously expensive and is far from complete. Not only all keys and certificates on millions of servers have to be renewed, also all secondary key material is affected: for example all user passwords must be considered compromised and therefore have to be replaced. Venafi [16] estimates that in April 2015, one year after the disclosure of Heartbleed, 74% of Global 2000 organisations are still exposed to attacks due to incomplete remediation. This is only 2% less then in August 2014!

Bugs in software come and go, but Heartbleed is unique in many ways: Never has a software bug has left so many private keys and other secrets exposed to the Internet for such a long time so easy to exploit without leaving any trace.

5.2. The Heartbeat Protocol Extension

The bug’s name “Heartbleed” derives from *Heartbeat* a protocol extension to the TLS/DTLS protocol introduced in January 2012 as defined in RFC 6520 [17]. It allows a SSL/TLS client to test if a remote connection is still alive without disturbing the data stream. The client sends an `TLS1_HB_REQUEST` with a nonce *payload string*. The server reads this string and echoes it back to the client. The client finally compares its own sent nonce *payload*

string with the received *payload string* from the sever. If they are equal, the connection is still alive.

The `TLS1_HB_REQUEST` package sent from the client contains a field called `payload` in the source code (Table 2, "Vulnerable Heartbeat code in C") that indicates the length of the *payload string*. This length is used to determine how many Bytes the server needs to copy in the `TLS1_HB_RESPONSE` package by executing the `memcpy`-function in line 6. Figure 2, "TLS Heartbeat protocol" shows how the server assembles a regular `TSL1_HB_RESPONSE` package based on the received `TSL1_HB_REQUEST` package. In order to simplify the graphic, the random padding of 16 Bytes which are systematically appended to both packages is not shown.

The code in Table 2, "Vulnerable Heartbeat code in C" implements mainly two functions: the lines 1, 2 and 6 *deserialise* the incoming `TLS1_HB_REQUEST` package and the lines 3 to 7 *serialise* the outgoing `TLS1_HB_REQUEST` package.

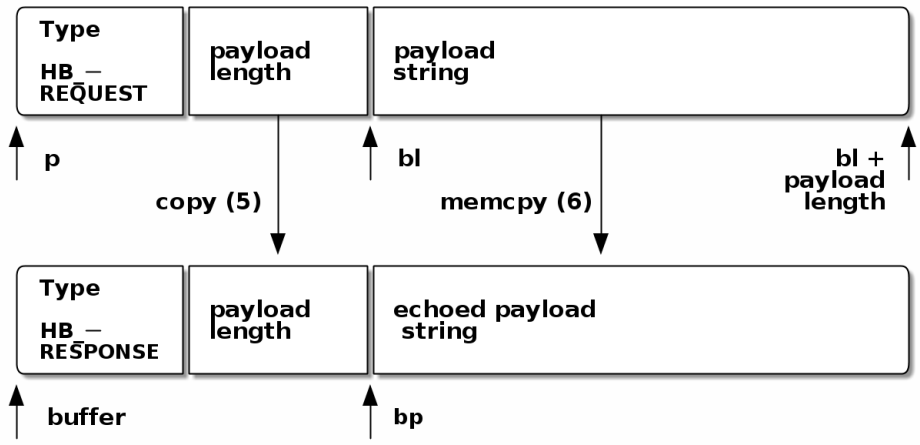

Figure 2. TLS Heartbeat protocol

5.3. How Does the Heartbleed-Exploit Work?

The so called *Heartbleed* vulnerability is an implementation problem, i. e. programming mistake in the very popular OpenSSL library [18] resulting from improper input validation due to a missing bounds check. It is important to note that there is no design flaw in the Heartbeat protocol itself.

The programming mistake was introduced with the first version of the Heartbeat feature committed in January 2012. Table 2, "Vulnerable Heart-

beat code in C" shows the details of the vulnerable Heartbeat-commit introducing the so called Heartbleed-vulnerability.

Exploiting the Heartbleed-vulnerability is very simple: the attacker sends a `TLS1_HB_REQUEST`-package with a spoofed `payload`-length field as shown in Figure 3, "TLS Heartbeat with spoofed payload-length-field". Here the `payload`-length needs to be larger than the actual *payload string* for example `0xffff`. OpenSSL receives the flawed package and deserialises it (cf. lines 1, 2 and 6 in Table 2, "Vulnerable Heartbeat code in C"). Because of a missing validity check the spoofed deserialised `payload`-length (cf. line2) is passed directly to the `memcpy(bp, pl, payload)` instruction (line 6) and is there used to determine how many Bytes are to be copied and sent back to the attacker. Thus a *Buffer-Overread* occurs and discloses 64KiB of internal OpenSSL memory.

From the functional point of view Heartbleed is best classified as a *CWE-502: Deserialization of Untrusted Data* weakness: "The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid."

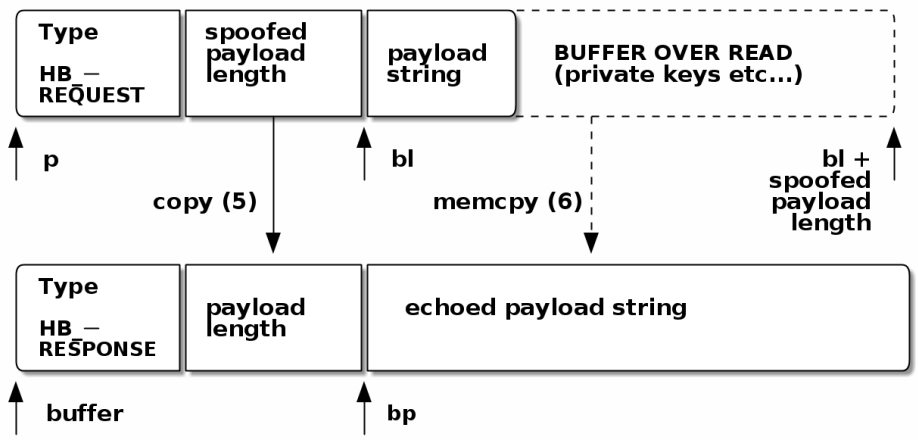

Figure 3. TLS Heartbeat with spoofed payload-length-field

Table 2. Vulnerable Heartbeat code in C

SHA	4817504d069b4c5082161b02a22116ad75f822b1
Author	Dr. Stephen Henson (Sun 01 Jan 2012 00:59:57 EET)
Com-miter	Dr. Stephen Henson (Sun 01 Jan 2012 00:59:57 EET)

Subject	PR: 2658 Submitted by: Robin Seggelmann Reviewed by: steve

```
unsigned char *p = &s->s3->rrec.data[0], *pl;
unsigned short hbtype;
unsigned int payload;
unsigned int padding = 16;

hbtype = *p++;      ❶
n2s(p, payload);    ❷
pl = p;


if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
        int r;

        buffer = OPENSSL_malloc(1 + 2 + payload + padding); ❸
        bp = buffer;

        *bp++ = TLS1_HB_RESPONSE;    ❹
        s2n(payload, bp);            ❺
        memcpy(bp, pl, payload);     ❻

        r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
    3 + payload + padding);  ❼
```

❶ Read type Byte from `TLS Request`.

❷ Read `payload`-length-field form `TSL Request`. **Note that there is no previous check if the actual payload string in `TLS Request` is as long as the `payload`-length field pretends it is!**

❸ Allocate memory for the `TLS Response`.

❹ Write the type in the response buffer.

❺ Serialise the `payload`-length into the response buffer.

❻ Copy `payload`-length number of Bytes from the request buffer into the response buffer.
→ **Buffer over-read vulnerability!**

Source: [18, `openssl-1.0.1/ssl_t1_lib.c`, lines=2405-2468]

5.4. The Heartbleed-Patch

Figure 4, “Heartbleed patch” shows how trivially the bug can be fixed. The patch introduces the missing validity check comprising two `if` statements.

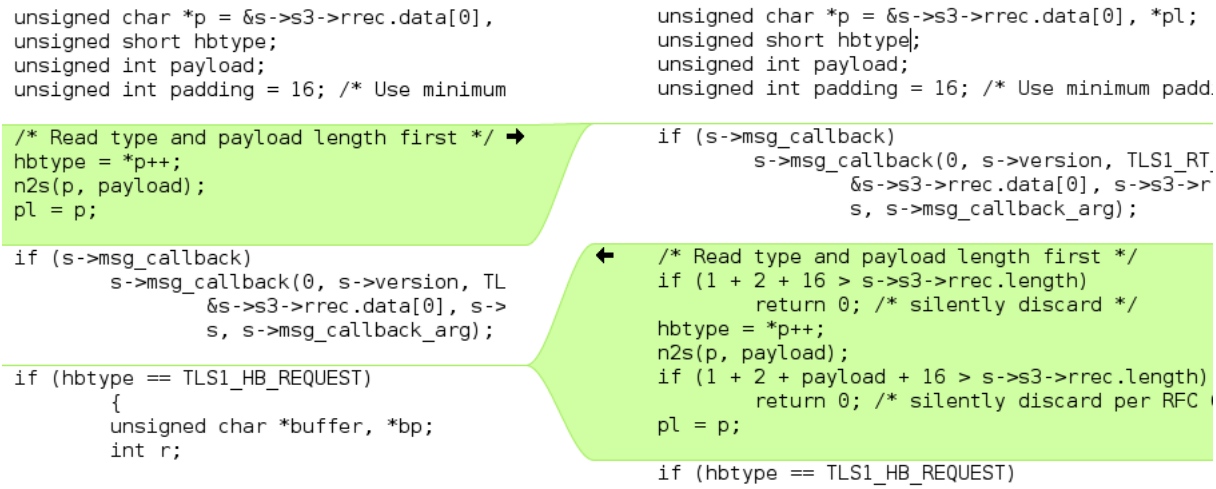

Figure 4. Heartbleed patch

The bugfix as shown in Table 3, “Patched Heartbeat code in C” in detail: The first `if` statement checks if the incoming `TLS1_HB_REQUEST` is as least as long as the its minimum required length and the second `if` statement checks if the *payload string* is at least a long the `payload`-length field pretends.

Table 3. Patched Heartbeat code in C

SHA	96db9023b881d7cd9f379b0c154650d6c108e9a3
Author	Dr. Stephen Henson (Sun 06 Apr 2014 02:51:06 EEST)
Committer	Dr. Stephen Henson (Mon 07 Apr 2014 19:53:31 EEST)
Subject	Add heartbeat extension bounds check.

```
if (1 + 2 + 16 > s->s3->rrec.length)                ❶
 return 0;    ❷
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)  ❸
 return 0;    ❹
pl = p;
```

❶ Is the `TLS Request` message is too short?
❷ If it is too short, then discard the request and do not answer.
❸ Is the `TLS Request` payload string at least as long as announced by the `payload`-length-field?
❹ If not, then silently discard the request according to RFC 6520 section 4.

Source: [18, `openssl-1.0.1g/ssl_t1_lib.c`, lines=2597-2603]

5.5. What Can We Learn?

The author of the code who introduced the Heartbleed vulnerability in OpenSSL is Dr. Seggelmann, a programmer who worked on the OpenSSL project during his PhD studies. Interviewed after the incident by the Guardian he said [19]: “I am responsible for the error, because I wrote the code and missed the necessary validation by an oversight. Unfortunately, this mistake also slipped through the review process and therefore made its way into the released version.”

For me there is no doubting the sincerity of this statement. But in view of future similar incidents, it is important to note that Heartbleed is part of a vulnerability class predestined for intentionally introduced weaknesses. As we saw in the previous chapter, Heartbleed can be classified as a “CWE-502: Deserialisation of Untrusted Data” weakness. This kind of vulnerability is listed as potential “CWE-505: Intentional Introduced Weakness” (cf. Figure 5, “Development Concepts” [21]).

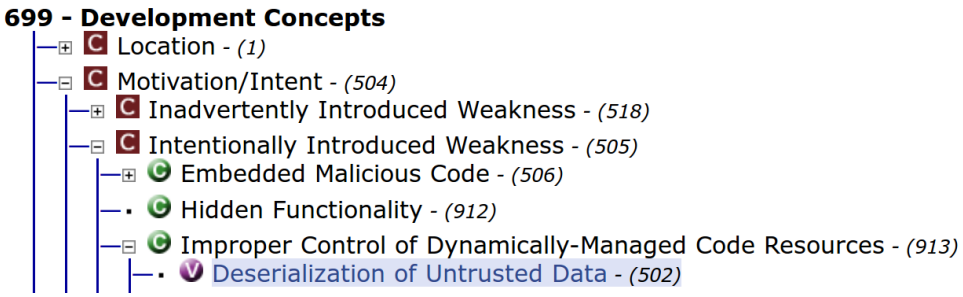

Figure 5. Development Concepts

Knowing about the efforts some public authorities undertook in weakening international crypto-standards, the *Sydney Morning Herald* asked shortly after the disclosure of Heartbleed [20]: “Is this a man who would purposefully leave a gaping hole in the internet, which the US National Security Agency could have been exploiting to spy on people’s communications?”. Dr Seggelman denied this in an interview with Fairfax Media. He said [20]:

“It’s tempting to assume that, after the disclosure of the spying activities of the NSA and other agencies, but in this case it was a simple programming error in a new feature, which unfortunately occurred in a security-relevant area.”.

In October 2015, the Dutch government donated €500,000 to the OpenSSL project to enforce Internet encryptions standards [22]. In a public statement, Mr. van der Steur, Holland’s Minister of Security and Justice, said that “his country opposes the idea of backdoors in encryption technologies, considering strong encryption vital for the protection of privacy for citizens, companies, the government, and the entire Dutch economy” [23].

One could interpret this donation as a hint that Heartbleed had been intentionally introduced. But we should not forget that a missing validation is a very common mistake. Furthermore, finding some missing lines of code is always harder than detecting a flaw in code. This is also true for code reviewers. Hence it is not surprising that the vulnerable code passed the review process. In fact, there is no explicit flaw in the vulnerable code: i.e. there are no explicit semantics in *C/C++* telling that `memcpy(bp, pl, payload)` can eventually mean something like: “copy the payload string including all secret private keys and certificates stored in memory”!

In *C/C++* many things may happen behind the scenes. This makes the language so elegant. But going back to the 1970s when *C* was developed: the systems had not yet had to deal with malicious input data. In those days, the main concern in writing computer programs was producing correct results for a small set of valid input data: *Do what is written in your program* can be well expressed in *C/C++* semantics. But the situation has fundamentally changed since. Once connected to the Internet, evil lurks everywhere. Today’s programs have to deal with every imaginable input. This is why we need programming languages that allow us to express: *Do only what is explicitly written in your program and **do nothing else**! - Rust* is such a language [24].

6. Could Heartbleed Have Happened With Rust?

The code in Table 4, “Vulnerable Heartbeat code in Rust” is a reproduction of the original C code (cf. Table 2, “Vulnerable Heartbeat code in C”) in *Rust* using the same variable names and function signatures, also without any

boundary checks. It allows us to observe how a *Rust* implementation would have dealt with the missing boundary checks leading to the Heartbleed vulnerability.

Table 4. Vulnerable Heartbeat code in Rust

```
fn tls1_process_heartbeat (s: Ssl) -> Result<(), isize> {
    const PADDING: usize = 16;

    let p = s.s3.rrec;
    let hbtype:u8 = p[0];
    let payload:usize = ((p[1] as usize) << 8) + p[2] as usize; ❶

    let mut buffer: Vec<u8> = Vec::with_capacity(1+2+payload+PADDING);
    buffer.push(TLS1_HB_RESPONSE);
    buffer.extend(p[1..1+2].iter().cloned()); ❷
    buffer.extend(p[3..3+payload].iter().cloned()); ❸

    let mut rng = rand::thread_rng(); ❹
    buffer.extend( (0..PADDING).map(|_|rng.gen::<u8>())
   .collect::<Vec<u8>>() );

    if hbtype == TLS1_HB_REQUEST {
        let r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, &*buffer);
        return r
    }
    Ok(())
}
```

❶ Extract the `payload`-length in big endian format from `TLS1_HB_REQUEST` package.

❷ Serialise _`payload`-length into `TLS1_HB_RESPONSE`.

❸ Copy *payload string*. **Note there has been no previous boundary check which leads to the Heartbleed vulnerability!**

❹ Append `PADDING` number of random Bytes. The padding has to be random here to mitigate certain cryptoanalysis attacks.

How does the vulnerable *Rust* implementation Table 4, “Vulnerable Heartbeat code in Rust” respond to Heartbleed exploits? Table 5, “Simulation of Heartbleed exploit package” simulates a Heartbleed attack with a malicious package. First the variable `payload` is set to `0x0101 = 257`; then line 3 of Table 4, “Vulnerable Heartbeat code in Rust” is fed with this input:

```
   buffer.extend(p[3..3+payload].iter().cloned());
```

`3+payload` is the upper index of the slice to be copied. Since the `Rrec` has only 22 elements, a `payload`-length of `257` is out of bounds. In the original C code this line caused the dreaded *Heartbleed* vulnerability with a *Buffer-Over-Read*. Table 6, “System response after Heartbleed attack” shows the response of the *Rust* code: The program aborts with a `panic` message. Thus the attacker is still able to trigger a deny of service attack but no *Buffer-Over-Read* occurs and no data is leaked!

Table 5. Simulation of Heartbleed exploit package

```
    let s: Ssl = Ssl {
        s3 : Rrec{
                rrec:  &[TLS1_HB_REQUEST, 1, 1, 14, 15, 16, 17,
                         18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
                         28, 29, 30, 31, 32]
        }
    };
    tls1_process_heartbeat(s).unwrap();
    }
```

Table 6. System response after Heartbleed attack

```
thread '<main>' panicked at 'assertion failed: index.end <= self.len()',
Process didn't exit successfully: `target/release/heartbeat` (exit code:
 101)
```

Like in the original patched *C* code Table 3, “Patched Heartbeat code in C” the critical out-of-bounds conditions can be easily silently discarded with two additional `if` clauses as shown in Table 7, “Patched Heartbeat code in Rust”.

Table 7. Patched Heartbeat code in Rust

```
fn tls1_process_heartbeat (s: Ssl) -> Result<(), isize> {
    const PADDING: usize = 16;

    if 1 + 2 + 16  >  s.s3.rrec.len() {return Ok(()) }          ❶
```

```
    let p = s.s3.rrec;
    let hbtype:u8 = p[0];
    let payload:usize = ((p[1] as usize) << 8) + p[2] as usize;
    if 1 + 2 + payload + 16  >  s.s3.rrec.len() {return Ok(()) } ❷

    let mut buffer: Vec<u8> = Vec::with_capacity(1+2+payload+PADDING);
    buffer.push(TLS1_HB_RESPONSE);
    buffer.extend(p[1..1+2].iter().cloned());
    buffer.extend(p[3..3+payload].iter().cloned());

    let mut rng = rand::thread_rng();
    buffer.extend( (0..PADDING).map(|_|rng.gen::<u8>())
   .collect::<Vec<u8>>() );

    if hbtype == TLS1_HB_REQUEST {
        let r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, &*buffer);
        return r
    }
    Ok(())
}
```

❶ Is the `TLS Request` message too short? If it is too short, then silently discard the request.

❷ Is the `TLS Request` payload string at least as long as announced by the `payload`-length-field? If not, then silently discard the request according to RFC 6520 section 4.

6.1. Results

The Heartbleed bug would not have been possible if OpenSSL had been implemented in *Rust*. We have seen that *Rust*'s memory management prevents inter alia *Buffer-Over-Read* which is the origin of the Heartbleed vulnerability. With *Rust* no disastrous data leakage could have occurred as it is impossible to read beyond any data structure. Notwithstanding this, attempts to read data out of bounds ends the process by default. This means in a Heartbleed attack scenario the impact would have been limited to a *deny of service*. From a theoretical point of view the findings are not surprising as they result directly from *Rust*'s core property: memory safety.

6.2. Suggested Improvements

In order to keep the *Rust* code in Table 7, “Patched Heartbeat code in Rust” comparable to the original in C Table 3, “Patched Heartbeat code in C” no optimisations were made. Here are some suggestions for improvement:

- For implementing complex protocols like SSL/TLS the Rust’s serialisation framework should be considered.
- Replace the `Vec` data structure with `Smallvec` that can be kept on the stack.
- The serialisation of `TLS1_HB_RESPONSE` should be outsourced in a proper function returning `buffer` and allowing automated tests.
- Replace return codes with own error `enum` types.

7. Rust on Embedded Systems

Of particular interest when dealing with embedded systems are *Rust*'s: *Guaranteed Memory Safety*, *Zero-Cost Abstractions* and *Iterators*.

7.1. Guaranteed Memory Safety

Rust's main innovation is the introduction of new semantics defining *ownership* and *borrowing*. They translate to the following set of rules which Rust’s type system enforces:

1. All resources (e.g. variables, vectors…) have a clear owner.
2. Others can borrow from the owner.
3. Owner cannot free or mutate the resource while it is borrowed.

By observing the above rules *Rust* regulates how resources are shared within different scopes. Memory problems can only occur when a resource is referenced by multiple pointers (aliasing) and when it is mutable at the same time. In contrast to other languages, *Rust*'s semantics allow the type system to ensure *at compile time* that simultaneous aliasing and mutation can never happen. As the check is performed at compile-time no run-time code is necessary. Furthermore, *Rust* does not need a garbage collector: when owned data goes out of scope it is immediately destroyed.

Table 8. Ressource sharing in Rust

Resource sharing type	**Aliasing**	**Mutation**	**Example**
move ownership	no	yes	`let a = b`
shared borrow	yes	no	`let a = &b`
mutable borrow	no	yes	`let a = &mut b;`

7.2. Iterators

A very common group of programming mistakes is related to improper handling of indexes especially in loops, e.g. “CWE-129: Improper Validation of Array Index” (cf. Table 9, “Common weaknesses in C/C++ affecting memory avoidable with iterators”).

Table 9. Common weaknesses in C/C++ affecting memory avoidable with iterators

CWE ID	Name
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
125	Out-of-bounds Read
129	Improper Validation of Array Index

In addition to traditional imperative loop control structures, *Rust* offers efficient iteration with functional style iterators. Like in Haskel iterators are lazy and avoid allocating memory for intermediate structures (you allocate just when you call `.collect()`).

Besides performance considerations, iterators considerably enhance the robustness and safety of programs. They enable the programmer to formulate loop control structures and to manipulate vectors without indexes! (See example in Table 10, “Vigenère cipher in Rust”).

Table 10. Vigenère cipher in Rust

```
fet p: Vec<u8> = s.into_bytes();   //plaintext
let mut c: Vec<u8> = vec![];    //ciphertext
```

```
for (cypherb, keyb) in p.iter()
    .zip(  key.iter().cycle().take(p.len())  ) {
        c.push(*cypherb ^ *keyb as u8);
    }
```

It must be noted that even with iterators *out of bounds*-errors may occur. Nevertheless, they should be preferred because they reduce the probability of errors related to indexes drastically.

7.3. Zero-Cost Abstractions

It is the language design goal *Zero-Cost Abstractions* that makes the *C/C++* language so efficient and suitable for system programming. It means that libraries implementing abstractions, e.g. vectors and strings, must be designed in a way that the compiled binary is as efficient as if the program had been written in Assembly. This is best illustrated with memory layouts: Figure 6, “Memory layout of a Rust vector” shows a vector in *Rust*. Its memory layout is very similar is to a vector in *C/C++*.

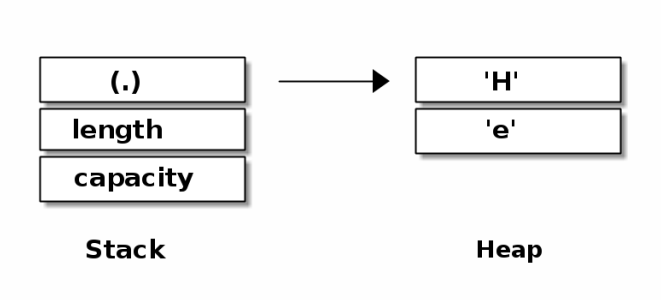

Figure 6. Memory layout of a Rust vector

Java, also a memory safe language, enforces a uniform internal representation of data. Here a vector has 2 indirections instead of 1 compared to *Rust* and *C/C++* (cf. Figure 7, “Memory layout of a Java vector”). As the data could be represented in a more efficient way in memory, we see that *Java* does not prioritise the *Zero-Cost-Abstraction* goal.

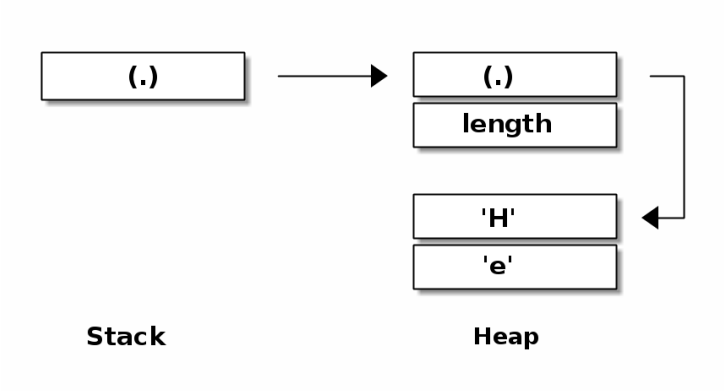

Figure 7. Memory layout of a Java vector

How about other data structures and overall performance? A good estimation is to compare benchmarks of small and simple programs. Too complex programs should be avoided for this purpose because variations of the programmer's skills may bias the result. According to the "Computer Language Benchmark Game" [25] *Rust* and *C/C++* have similar benchmark results.

As *Rust* uses the LLVM framework as backend, it is available for most embedded platforms.

7.4. Final evaluation

The above discussed features are only a small selection of *Rust*'s qualities. Not less important features like *threads without data races*, *minimal runtime* and *efficient C bindings* result together in an ideal tool for secure embedded system programming.

References

[1] D. Weinstein, 'CVE-2015-0096 issue patched today involves failed Stuxnet fix', *Hewlett Packard Enterprise Community*, 03-Oct-2015. [Online]. Available: http://community.hpe.com/t5/Security-Research/CVE-2015-0096-issue-patched-today-involves-failed-Stuxnet-fix/ba-p/6718402. [Accessed: 02-Jan-2016].

[2] B. Schneier, 'The Internet of Things Is Wildly Insecure — And Often Unpatchable', *Wired online magazine*, 06-Jan-2014. [Online]. Available: http://www.wired.com/2014/01/theres-no-good-way-to-

patch-the-internet-of-things-and-thats-a-huge-problem/. [Accessed: 02-Nov-2015].

[3] R. Eikenberg, 'Fatale Sicherheitslücken in Zwangsroutern von Vodafone/Kabel Deutschland', *Heise Security*, 30-Oct-2015. [Online]. Available: http://www.heise.de/newsticker/meldung/Fatale-Sicherheitsluecken-in-Zwangsroutern-von-Vodafone-Kabel-Deutschland-2866037.html. [Accessed: 02-Nov-2015].

[4] D. Bongard, 'Offline bruteforce attack on WiFi Protected Setup', presented at the Passwordscon 2014, Las Vegas, 2014.

[5] A. Greenberg, 'Hackers Can Disable a Sniper Rifle—Or Change Its Target', *Wired online magazine*, 29-Jul-2015. [Online]. Available: http://www.wired.com/2015/07/hackers-can-disable-sniper-rifleor-change-target/. [Accessed: 02-Nov-2015].

[6] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, 'Basic concepts and taxonomy of dependable and secure computing', *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, no. 1, pp. 11–33, 2004.

[8] VDC Research Group, Inc., 'An A+ for C++ - On Target: Embedded Systems', *IoT & Embedded Software Development*, 09-Apr-2012. [Online]. Available: http://blog.vdcresearch.com/embedded_sw/2012/09/an-a-for-c.html. [Accessed: 04-Jan-2016].

[9] L. Szekeres, M. Payer, T. Wei, and D. Song, 'Sok: Eternal war in memory', presented at the Security and Privacy (SP), 2013 IEEE Symposium on, 2013, pp. 48–62.

[10] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, and M. Payer, 'The matter of Heartbleed', presented at the Proceedings of the 2014 Conference on Internet Measurement Conference, 2014, pp. 475–488.

[11] P. Mutton, ‘Half a million widely trusted websites vulnerable to Heartbleed bug’, *Netcraft*, 08-Apr-2014. [Online]. Available: http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html. [Accessed: 21-Dec-2015].

[12] Codenomicon, ‘Heartbleed Bug’, 29-Apr-2014. [Online]. Available: http://heartbleed.com/. [Accessed: 23-Dec-2015].

[13] M. Riley, ‘Nsa said to exploit heartbleed bug for intelligence for years’, Bloomberg News, April, vol. 12, Apr. 2014.

[14] M. Riley and J. Robertson, ‘Why Would Chinese Hackers Steal Millions of Medical Records?’, *Bloomerg Business*, 18-Aug-2014. [Online]. Available: http://www.bloomberg.com/news/2014-08-18/why-would-chinese-hackers-steal-millions-of-medical-records-.html. [Accessed: 05-Jan-2016].

[15] K. Bocek, ‘How an Attack by a Cyber-espionage Operator Bypassed Security Controls’, *Venafi*, 28-Jan-2015. [Online]. Available: https://www.venafi.com/blog/post/infographic-cyber-espionage-operator-bypassed-security-controls/. [Accessed: 21-Dec-2015].

[16] Venafi Labs Analysis, ‘Hearts Continue to Bleed: Heartbleed One Year Later’, *Venafi*, 08-Apr-2015. [Online]. Available: https://www.venafi.com/assets/pdf/wp/Hearts-Continue-to-Bleed-Research-Report.pdf.

[17] M. Tuexen, R. Seggelmann, and M. Williams, ‘Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension’, Internet Engineering Task Force (IETF), no. RFC 6520, Feb. 2012.

[18] ‘OpenSSL: Cryptography and SSL/TLS Toolkit’. [Online]. Available: https://www.openssl.org/source/. [Accessed: 25-Dec-2015].

[19] A. Hern, ‘Heartbleed: developer who introduced the error regrets “oversight”’, *the Guardian*, 11-Apr-2014. [Online]. Available: http://www.theguardian.com/technology/2014/apr/11/heartbleed-developer-error-regrets-oversight. [Accessed: 23-Dec-2015].

[20] L. Timson, ‘Who is Robin Seggelmann and did his Heartbleed break the internet?’, *The Sydney Morning Herald*, 11 Apr. 2014.

[21] CWE, ‘CWE - Common Weakness Enumeration’, *MITRE Corporation*. [Online]. Available: https://cwe.mitre.org/. [Accessed: 06-Jan-2016].

[22] ‘34 300 XIII Vaststelling van de begrotingsstaten van het Ministerie van Economische Zaken (XIII) en het Diergezondheidsfonds (F) voor het jaar 2016’, *Tweede Kamer der Staten-Generaal*, vol. 34 300 XIII, no. 10, p. 2, Oct. 2015.

[23] C. Cimpanu, ‘Dutch Government Donates €500,000 to OpenSSL, Publicly Opposes Encryption Backdoors’, *Softpedia*, 4 Jan. 2016.

[24] ‘The Rust Programming Language’. [Online]. Available: https://doc.rust-lang.org/book/. [Accessed: 17-Feb-2016].

[25] ‘C g versus Rust’, *The computer Language Benchmarks Game*. [Online]. Available: http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=gpp&lang2=rust. [Accessed: 07-Jan-2016].