Sins of the Fathers—Yesterday's Cybersecurity Today

In Lewis Caroll's *Through the Looking Glass*, the Red Queen takes Alice on a mad run. They run as fast as they can, but no matter how fast they run, they always stay in the same place. That is odd, thinks Alice, and she says so.

> 'In our country you'd generally get to somewhere else - if you ran very fast for a long time as we've been doing.'
>
> 'A slow sort of country!' said the Queen. 'Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!'

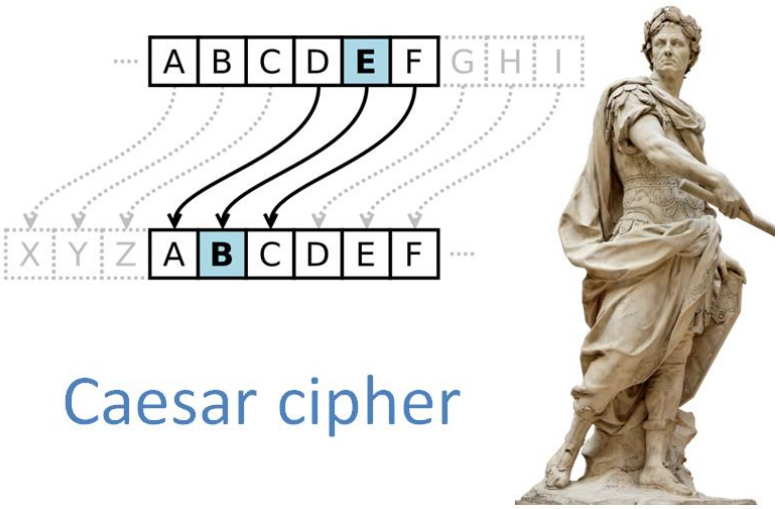
The Red Queen effect is typical for evolutionary arms races. In the course of millions of years, the ancestors of zebras and lions both evolved. Zebras became faster and better at seeing, hearing and smelling predators—useful, if you want to outrun the lions. But in the meantime, lions also became faster, bigger, stealthier and better camouflaged—useful, if you like zebra. So, although the lion and the zebra both "improved" their designs, neither became more successful at beating the other in the hunt. Lions and zebras are locked in an arms race. They are running to stand still.

The Red Queen effect also applies to cybersecurity. Attacks become ever more sophisticated to deal with increasingly advanced security measures. An arms race that may be impossible to win. Some see it as a lost cause. They throw up their arms and say that all is pointless and we are never going to win. I disagree. Yes, we may be running to stand still, but we need to keep running regardless. Just ask yourself: if it stopped running, what would happen to the zebra?

Cybersecurity is also the art of not ending up as someone's lunch. The question is: how?

Systems and Network Security

Traditionally, information security is about cryptography, sending messages in a secret code so that even if they should fall into the hands of the enemy, the enemy will not be able to read them. For instance, in every computer science department in the world, the first thing the students learn about security is typically the *Caesar cipher*.



It is one of the simplest and oldest encryption techniques and it was named after Julius Caesar who used it. The idea is very simple. Caesar replaced each letter in the message, for instance, with the letter that is three positions to the left in the alphabet. So a 'D' became an 'A', an 'E' became a 'B', and so on. It really is very simple to decipher as long as you know it, but if you do not, the message

looks like a random collection of letters. The Caesar Cipher is over 2000 years old, but when I recently used it in an exam, about one third of the students failed to decrypt it. Actually, what I had done, was encrypt part of the exam using the cipher, so the students would first have to decrypt their exam before they could even see the questions. I thought that was very clever and very funny. I did not yet realise that some students would also encrypt their answers. It took me a long time to grade these exams!

The point is that cryptography has been around for thousands of years, growing increasingly sophisticated, and with a strong tradition in the Netherlands. The trick is to hide the information in such a careful manner that it is impossible to decrypt the message without the key. With the key, however, getting the original message back is easy. If the enemy cracks the key, all secrecy is lost. This is what happened to the Germans in WWII when Alan Turing and his fellow cryptographers at Bletchley Park cracked the German Enigma code, which gave the British forces access to communication with German planes, U-Boats and army units. Clearly, cryptography is important.

In cryptography it is not unusual to start a lecture just like I just did: with Alice. But now Alice is a different fictional character that always wants to send a message to her friend Bob in a cryptographically secure way. So the story is always:

> "Alice wants to send a message to Bob"

What you do not hear very often is:

> "Alice wants to send a message to Bob but Bob was bitten by a zombie"

even though, you will agree, this would be a much more interesting story.

Moreover, most security incidents today are not caused by attackers breaking the encryption between Alice and Bob. Instead, the zombie story is very common. In security terms, a zombie is a computer that was compromised. For instance, the attackers use a vulnerability in your web browser to gain control over that program. Subsequently, they elevate their privileges until they have full control over your machine. They can use it for anything they want. Your machine is a zombie, or a bot in a network of compromised computers controlled by a cybercriminal. You are pwned!

Most security incidents today are caused not by broken cryptography, but by the sorry state of the software running on our computer systems. That software is rife with security holes.

As a researcher, I am interested in tough problems, but also in problems that matter. Problems that affect the lives of real people. While this is true for cybersecurity in general, cryptography not excluded, I believe that I can have most impact at what we call the system level: the problems that require deep knowledge of hardware and software, assembly language, and, yes, hacking. I want to develop new techniques to find vulnerabilities in software, to develop better systems, to harden existing programs, and, yes, to attack criminal infrastructures.

It is also what I know best, what I love. When I was 16, I was in a (rather pathetic) group of hackers. We called ourselves Babysoft and we cracked a few computer games—we disabled the copy protection, so that we could play them without paying, and give them to our friends, and so on. We were not very skilled, at least I was not, but it forced me to learn assembly language and look for weaknesses in software, and in general, to hack. Plus, we made up for our lack of skills, by leaving bizarre tags, or "crack intros". If you ever played one of "our" games, you may remember the scrolling "Cracked by Babysoft" message, followed by a declaration of love to the girls that we fancied. The idea was that they would see the crack intro and fall hopelessly in love with us. Interestingly, there were only 3 of us involved in the cracks, but the list of girls was much longer. I guess we were not very picky.

As you can imagine, we (Babysoft) struck fear into the heart of the computer industry—and even more fear into the hearts of the girls in our class. However, you will also agree that it was all fairly innocent: we did not make any money, we did not destroy machines of a nuclear facility, and we did not DDoS any banks. We were in it for the admiration of our friends, and would-be girlfriends. Now all that has changed. Cybercrime is serious.

There is a Red Queen at the root of this problem. She forces organisations to join the "digital revolution", a mad run to connect everything to the Internet—because if *you* don't, your competitor will and you will be out of business. As a result, programs in governments, banks, hospitals, powerplants, and shops, that were once isolated, are now vulnerable to cyberattacks from anywhere on the planet.

In the Netherlands, as in the rest of Europe, our academic fathers have traditionally been very strong at the mathematical and theoretical side of information security, but much less so in the system-level aspects—hacking. I believe that the time has come to focus more on *systems*.

For instance, what interests me, is how they do it. How do the attackers break into our laptops, phones, routers, and nuclear installations and why are we unable to stop them? How do they construct their criminal infrastructures? And how can we protect our systems better?

The Bugs

The reason attackers can break into our systems is that they are buggy. In practice, all software has bugs. Even very good programmers introduce about two bugs per thousand lines of program code. That may not sound like much, until you realize that operating systems like Windows, Mac OS X, and Linux already count tens of millions of lines of code, or tens of thousands of bugs. Then there are the applications: another few millions of lines of code, and thousands of bugs. Some of these bugs are security bugs, or vulnerabilities.

It gets worse. To make a system really secure, we need to find and fix every single vulnerability. An attacker, on the other hand, needs to find only one. This is a well-known asymmetry between attackers and defenders that works in the attacker's favour. What I want to add is that we also should find and fix every vulnerability for all possible attacks—past, present, and future. Software tends to stay in use for a long time. A program that may be well protected against all threats known today, may be hopelessly vulnerable tomorrow if attackers discover a new kind of exploit. In 1988, when the first Internet worm —in a catastrophic outbreak that swept across the Internet—showed that certain programs really are vulnerable to buffer overflows, you may have been pretty secure if you guarded your program against such attacks, but that is only because at that time attackers had not yet discovered other attack vectors, like dangling pointers, or format string bugs.

Legacy software is everywhere: in security systems, industrial control systems, embedded in lifts, vehicles, traffic control, train signalling, and so on. Software that was designed by a previous generation and still works perfectly, except that it was not built for the security threats of today. Perhaps the original programmers, our fathers and the fathers of these programs, have long since retired, but the code is still there. Perhaps we no longer even know how it works. Perhaps we no longer even have the source. But we still need to care.

The fundamental issue is this: how could the developers have known what to guard against, if the problems had not been invented yet? What should we guard against today to feel safe tomorrow?

Few, old, dumb—and dangerous

Let us have another look at the bugs. I just mentioned that there are thousands of them on every system and more each year. Now I will argue the opposite. There really are not that many security bugs at all! At least, the number of different *kinds* of bug at the system level is quite small. A few

handfuls, depending on how you count. Buffer overflows, dangling pointers, format string bugs, SQL injection—sure there are others, but the point is that there are not that many.

Moreover, most of these vulnerabilities are old news. Buffer overflows, for instance, are a top-3 threat today and have been a top-3 threat for as long as I remember. Conceptually, they are really simple. A program forgets to check whether the data it receives actually fit in the memory reserved for them. If an attacker sends too many bytes, well, they have to go somewhere, so the program just keeps writing them, overwriting adjacent memory—just like when you are supposed to fill only one cube in an ice cube tray, but pour in too much water, and the water overflows the adjacent blocks.

Whatever was originally in that memory is overwritten; it is gone. So if it contains important information, like your bank account number, or your authorization level, or the program to execute next, an attacker can now overwrite it and change it: the bank account number, the authorization level, or the program to execute next.

That is it. It really is that simple.

What is astonishing is that this very simple security bug was already used over a quarter of a century ago in that world's first computer worm I mentioned earlier—an attack that essentially brought the Internet to its knees. In fact, memory corruptions bugs were already discussed in a study by the US Air Force as early as 1972. At that time, there was no Microsoft yet. No Apple. No Linux. The Beatles had just split up! Clearly, this is old school. And yet, the buffer overflow was still used in Operation Red October, a cyber espionage program, discovered earlier this year, that had been operational undetected for years, stealing information from diplomats and government officials worldwide. So it goes.

From bringing down the Internet to spying on diplomats, the costs of these bugs have been extraordinary—why haven't we been able to stop them? Why are they still a top-3 threat?

Sins of the Fathers?

Probably without exception, the common attack vectors today are the result of design decisions in the past. For instance, much of the system software in use today was written in the C programming language (and languages closely related to it, like C++ and objective C). Dennis Ritchie is known as the father of the C programming language. When he designed the language in the early 1970s at AT&T, he did so because he needed a programming language for the new operating system he was building with Ken Thompson, which became known as UNIX. To me, C is just beautiful: fast, elegant, and extremely concise. Not everyone agrees. Even so, the C language and its cousins are still the most used programming languages in the world. And they are inherently vulnerable to buffer overflows.

Indeed, buffer overflows are normal behaviour for C programs. Standard functions from the C library (for the insiders: functions like *memset()* and *memcpy()*), encourage programmers to overwrite entire

areas of memory, without respecting the boundary of buffers they may contain. This is very efficient, but you do not even have to squint very hard to see that doing so will overflow all the buffers inside.

C and C-like languages have more problems. They have fancy names that are only meaningful for insiders. Dangling pointers. Format string bugs. Null pointer dereferences. The point is that the number of ways a programmer can goof up in C is impressively long.

Then why use it? There are other languages that do not suffer from these vulnerabilities—should we not simply get rid of all that C code and use safer languages, like Java, or C#? I once attended a seminar where a member in the audience exclaimed—almost angrily:

> "These are not just bugs, but crimes. With so many better languages available, there is no excuse for them today!"

Well, wake up!

First, it is not reasonable to ask the computer industry to throw away everything it has done and start over again. It is like asking the zebra to become an elephant because then it would be less vulnerable to lions. C is the most popular language today, not because it so beautiful, but because it was the most popular language in the past. Much of our software was written in C. We are talking about enormous investments. Reimplementing an existing code base is very costly and typically introduces many new bugs. If you do this as a company, you will be slower to get your products to market, lose money, and risk being swept away by the competition.

This is another example of the Red Queen effect. Why are trees in a forest so tall? Because all other trees are tall. No single tree can afford not to be, even though it is costly to put so much energy is growing. I am no economist, but to me the market works the same way. There is little time to slow down, to redesign, and "do things properly", because we have to keep up with the others. And since the others are doing the same, we are all running to stand still relative to each other.

Second, and this is more important, the problems do not go away just because you use a different language. System-level bugs occur because of bad protocols, bad runtimes, bad interfaces, design errors, race conditions, interactions with the operating system, and so on. Today, attackers also use JavaScript. Similarly, Java has become a popular target for attackers—to the point that earlier this year the US government recommended disabling it. Famously, a Java vulnerability was used in the Red October cyber espionage program. The attack surface is enormous. Hackers are inventive.

Still, I like the idea of thinking of certain bugs as crimes, or as I would have it: *sins*. Just like a car should not explode when you bump it from behind, or the batteries in a laptop catch fire, there are certain security bugs that software today simply should not have. While not my area at all, I would be very interested in research that looked into liability as a security measure, i.e., to what extent commercial software vendors can be made liable for security incidents caused by preventable security bugs in their code.

The main thing I want to stress, however, is that programmers make mistakes. Some of the mistakes that lead to security vulnerabilities while dumb, are partially due to the tools we put in the programmers' hands: systems and languages that were designed in a different age, with less attention to security. Tools that make it easy to make mistakes. Because of decisions made long ago.

And however careful you design programs in the future, there is much legacy software. Billions of lines of code, riddled with bugs, full of vulnerabilities. Perhaps designed without security in mind, or without knowledge of the latest exploitation techniques. We cannot ignore these programs, pretend they do not exist and design the future from scratch. They are there. They will not go away quickly.

Let us now switch gears and look at the consequences of these vulnerabilities.

Beauty on the dark side

It is fascinating what a skilled hacker can do with these simple bugs. For instance, it is possible to change the way the program executes. Doing so is complicated, but also very rewarding. The basic idea is not hard to understand.

One of the things a program stores in its memory is where, at which instruction, to continue executing after it is done with a particular function—for instance, after handling a client's request. This is no different from our own behaviour. When we get tired reading a book before going to bed, we insert a bookmark, so we remember the next night where to continue. The program does the same, but writes this information in its memory. Now suppose that the attackers overwrite exactly *this* information—perhaps using a buffer overflow. It means that they can make the program execute instructions that are different from what the program intended. Doing so repeatedly allows them to string together little bits of the original program code to build their own, new functionality—reusing the program's own instructions against it, like a strange parasite, feeding on the original code.

If it is true that software controls a computer just like a brain controls a body, then it really does start to look like a zombie infection eating computer's brains. It reminds me of the zombie ants found in tropical forests around the world. These black carpenter ants, as they are properly known, are just like any other ant, until they make the mistake of touching a particular fungus, which enters the ant's body and affects its brain. In short, it makes the ant get to the ground from the tree where it lives, walk up some plant's stem, bite into a leaf on the northern side of the plant and die there—leaving its body for the fungus. Likewise, many viruses manipulate a host cell's own resources to replicate. This is similar. The strange, parasitic code injected by the attacker makes the program do anything it wants to, including spreading the infection and turning off the system's protective measures.

The really clever thing the hacker has done is use the simple vulnerability, perhaps a buffer overflow, and construct what I can only describe as a 'weird machine'. It is a term that I did not invent, but wish I had, because it is so apt. A weird machine is like a higher-order machine constructed on top of the original system that allows attackers to execute programs different from the original code, to break free from the program's mold. The weird machine may have a very different (and very strange) instruction set and require a very unusual way of programming, but if you look at it from a distance and squint a little, it is exactly the same.

To construct a weird machine, the hacker must put the program in exactly the right state, providing the right inputs, making sure that the right bytes are in memory in the right places, and so on. It is an art. Again, there are different kinds of weird machines that hackers can construct, but not that many. Some are more suited for one attack, others for another.

Thus, this simple vulnerability offers a pathway to many abilities some consider to be… unnatural. Weird programs run in crazy machines, doing things that are well beyond anything the original programmer ever imagined. This is beautiful. I would like you to recognise the beauty of it.

Moreover, as observed by Sergey Bratus Of Dartmouth College and his colleagues, it brings us full circle to Alan Turing. Before he went on to crack the German Enigma code, Turing worked on the most fundamental issues in computer science, including what is computable with a given machine or program. We are now faced with this problem again: what can be computed with this program, including all the possible weird machines we may construct on top of it? Some weird machines are limited. Perhaps they allow you to read a value that is in memory, but not modify it, or you cannot run arbitrary functions. In the most general case, the weird machine is universal, or Turing complete, which informally means that given enough memory you can compute anything.

In fact, in my group we have developed new, generic and very stealthy ways of constructing a Turing complete weird machine. Yes, this is offensive research—taking the attackers' position. I mention this, because I want to discuss good research and bad research later. But let us first look at the criminals a little more.

Gaining control over a program running on your computer is only a first step. Attackers build on this and download further programs that complete the infection. These new programs install themselves in hidden ways and make your computer join a botnet, a collection of compromised machines, or zombies. These botnets may include millions of other zombies, all under the control of a criminal. They form the backbone of criminal activities, the core infrastructures that allow highly professional criminal organizations to steal money, leak private data, destroy information, or attack other systems. The infrastructures are specialized and mutually supportive. For instance, some infrastructures offer a pay-per-install service. You can pay these criminals to have your malware installed on other machines. Handy, if a cleanup operation just disinfected a part of your botnet. Other infrastructures specialize in theft, money laundering, DDoS attacks, and so on.

What interests me, is how powerful these infrastructures are. Can we still disrupt them? Not just now, but in five, or ten years. For this reason, my group infiltrates these infrastructures, studies them, analyzes how they evolve and what their weaknesses are. This is research that is exciting, interacting directly with criminal organizations, and clearly useful, but also a difficult balancing act in deciding what is still advisable, permissible and valid academic research. When in doubt, we consult other parties, including law enforcement agencies. And we have a role to engage with society in the public debate, for instance about the powers such law enforcement agencies should and should not have to deal with cybercrime.

Botnets have become amazingly resilient, amazingly powerful. The firepower of a few hundred thousand zombies—in a DDoS attack—is enough to disrupt all but the most resilient systems. In my group, we evaluate such firepower, see what the attackers *can* do, if they really mean business. Again, this is offensive research. We look for techniques that the attackers *could* employ to do even more damage. Likewise, we look for ways in which attackers could hide their malicious functionality in the future, weaknesses in current protection schemes.

So, let us now turn around and face that awkward question: should we even do this kind of research? What is good research? What is not?

Attack and Defense: Good and Bad

The Red Queen effect puts a damper on security research. We do this, they do that. We think of a clever new defense, they find clever ways to circumvent it. It is rare to find solutions that stand the test of time.

Within this arms race, researchers can either look for new defenses or for new attacks. With my team, I focus mostly on defenses, but it is not the case that defensive research is necessarily good and attack research bad. There is much bad research on protective measures. The essential criterion here should not be whether a solution detects or stops today's attacks, but whether it can be evaded with trivial changes to the attack. If so, it is not a solution; it is bad research.

This is not the case for attack research. Attacks are valuable, even if they can be prevented easily. Attack papers prove the seriousness of the vulnerability, provide insight in the underlying systems, and teach us the kind of mistakes to avoid in the future. Of course, we need to disclose the vulnerabilities in a responsible way, preferably with fixes. Moreover, for academic research it is not sufficient to simply break one system after another. Science generates knowledge by building on previous results. Thus, there should be lessons to learn. There is little to learn from a lion catching yet another zebra in the same old way, but it gets interesting if it thinks of a clever new trick to do so.

We also need attack research to target malicious systems, like botnets. Infiltrating these systems, analyzing them and stopping them, requires the skills of a hacker. We develop new techniques to reverse engineer and dismantle such systems. There is good news here. Many people seem to think that malicious software is developed by some sort of super-hackers, but this is typically not true at all. Phrased differently, their code is full of bugs too! We have found so many bugs in malicious software, we considered filing bug reports.

Coming from a systems background, I mostly like to build better and more secure systems. But even here, some of our techniques are also used by attackers and are sometimes referred to as 'the dark side of the force'. One example is my team's research on security testing. We want to find the security bugs before the attackers do—in the testing phase. One way to do so is to feed the program with random or invalid data and wait to see if anything unexpected happens. Unfortunately, you may have to wait a long time. The reason is that a vulnerability is often dependent on one very peculiar state of the program, while the total number of states in real software far exceeds the number of particles in the universe. Finding that one state that triggers the bug is not easy. This is why we still have those thousands of bugs I mentioned earlier. Our approach is to extract in advance as much information about the program as possible by means of reverse engineering. Using that information, we then decide which parts of the program are most likely to contain the bugs and test only those. For instance, you cannot have a buffer overflow without a buffer and so we focus our tests on the instructions that use the buffers, perhaps in a loop. Doing so reduces the state space tremendously. Better testing clearly improves the security of our software. On the other hand, automatically finding vulnerabilities—would that not be interesting for cybercriminals?

Moral dilemmas

These dilemmas occur constantly, and require careful consideration. When is it okay to do this research, when is it not? To me, it makes my field more interesting than most. Besides the grand challenges in the research, which, as I have shown, go back to fundamental questions and issues that have dogged computer science for decades, there are ethical questions that creep up from day one. The most important one is which side to be on. As a hacker you have skills that are very powerful. How will you use those powers? It really is just like Star Wars.

I want to work in this area—on the good side. With my group, I want to investigate the inherent vulnerabilities in our systems, the weird machines, and the fundamental questions that they raise. I also want to develop solutions to protect our information systems and build better new systems that are more secure from the outset.

To do so we need deep technical insight at the system level. As a society, we have done a bad job at fostering such knowledge, thinking perhaps that we could outsource system level coding and not worry too much about the bits and bytes that keep our systems running. The truth is, we lack people who understand the weaknesses of our systems, who can deal with attacks and malware, and who can fix our systems and build better ones. The truth is, we lack expertise to keep up in the race, never mind winning it. The truth is, we need hackers.

I have explained how many of our security problems are caused by design decisions in the past. Someone designs a program language or a system and years later someone else finds a design flaw that allows a hacker and all his malware to take over your system. Ideally, I want an UNDO button, a way to make that problem go away. A fix that prevents the attacker from abusing the vulnerability. Make him go back to square one, as if we had uninvented the hack. Not perfect security, but I want to go back to the situation that existed before the problem surfaced. Yesterday's security, today.

Acknowledgments

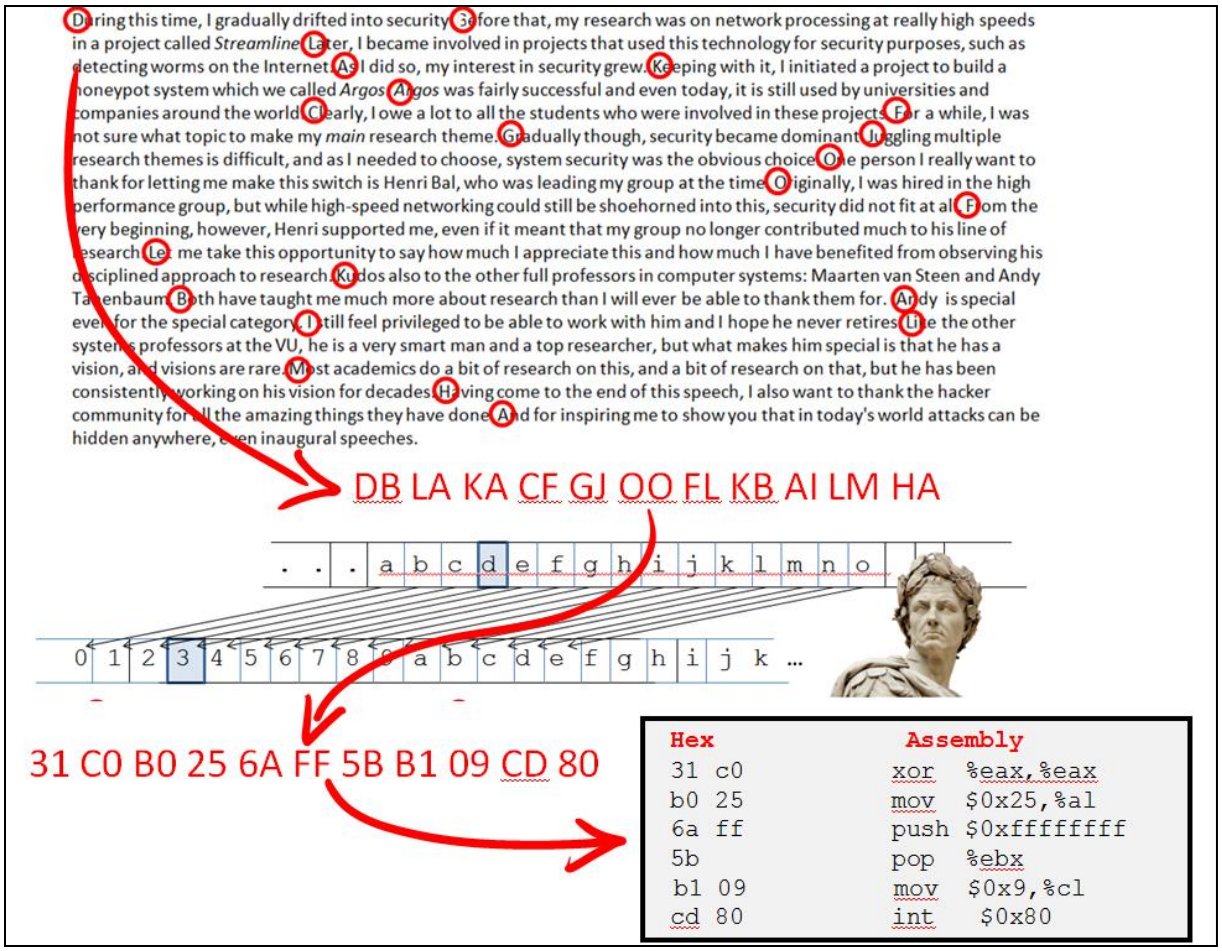
Now that my inaugural speech is drawing to an end, I would like to thank the people without whom I would not have stood here today, starting with my parents. Without them, I would not have stood here today. They taught me to be curious about everything. In light of the title of this speech, I want to single out my own father. He stepped down from politics, because doing so for him was better than compromise on his principles. I admire him greatly for that.

Most of all I want to thank Marieke, for her support and love, and for trying to make me a better person. Without her, who knows where I would be? I also would like to thank Duko and Jip, the two Jedi warriors who made me appreciate Star Wars.

In research, I would be nowhere either without the amazing people I have worked with in the past few years. I am well aware how lucky I have been to have had such exceptionally good students and researchers in my group. If there is anything I have done well, it is that I have found people around me who are smarter than I.

I am grateful to Harry Wijshoff, for giving me a chance in Leiden, and to the VU in general and Henri Bal in particular for making it possible for me to work in one of the best computer systems groups in Europe. It is hard to believe I have worked here for almost ten years already. It has been great.

During this time, I gradually drifted into security. Before that, my research was on network processing at really high speeds in a project called *Streamline*. Later, I became involved in projects that used this technology for security purposes, such as detecting worms on the Internet. As I did so, my interest in security grew. Keeping with it, I initiated a project to build a honeypot system which we called *Argos*. *Argos* was fairly successful and even today, it is still used by universities and companies around the world. Clearly, I owe a lot to all the students who were involved in these projects. For a while, I was not sure what topic to make my *main* research theme. Gradually though, security became dominant. Juggling multiple research themes is difficult, and as I needed to choose, system security was the obvious choice. One person I really want to thank for letting me make this switch is Henri Bal, who was leading my group at the time. Originally, I was hired in the high performance group, but while high-speed networking could still be shoehorned into this, security did not fit at all. From the very beginning, however, Henri supported me, even if it meant that my group no longer contributed much to his line of research. Let me take this opportunity to say how much I appreciate this and how much I have benefited from observing his disciplined approach to research. Kudos also to the other full professors in computer systems: Maarten van Steen and Andy Tanenbaum. Both have taught me much more about research than I will ever be able to thank them for. Andy is special even for the special category. I still feel privileged to be able to work with him and I hope he never retires. Like the other systems professors at the VU, he is a very smart man and a top researcher, but what makes him special is that he has a vision, and visions are rare. Most academics do a bit of research on this, and a bit of research on that, but he has been consistently working on his vision for decades. Having come to the end of this speech, I also want to thank the hacker community for all the amazing things they have done. And for inspiring me to show you that in today's world attacks can be hidden anywhere, even inaugural speeches.



You see, as we have reached the last paragraph of the last page, the time has come to tell you that the previous paragraph contains malicious code. Each of the begin letters of the sentences is a small part of a program, but in an encrypted form. Specifically, I encrypted the letters with a 2000 year old Caesar cipher. If we put them together and shift each letter 10 positions to the left, we get the real machine code of the attack. If we write it as assembly instructions, some of you will recognise it for what it is. On Linux PCs, it will kill all running programs. On that note, I will gracefully exit my program. May the force be with you.

Ik heb gezegd.