



CHECK POINT RESEARCH

# WHAT'S VULNERABILITY RESEARCH?

# WHO IS THIS ARTICLE FOR?

Anyone with at least a passing interest in the field of vulnerability research who's taken aback by the cloud of terms to memorize, processes to follow and names to know. No prior technical knowledge is required. This article doesn't teach actual vulnerability research past the very basics of the basics; if you're looking for a text that does, go read our own "[A First Introduction to Systems Exploitation](#)".

## Introduction

On my deathbed, I'm pretty sure that for an instant I will recall the first time I got to attend the Chaos Communication Congress.

CCC takes place every year in Germany, from December 27th to the 30th, and it is one of the largest information security conferences in the world; even back then, the number of attendees exceeded ten thousand. This was back in the old normal, when people were excited to catch airplanes and gather in large crowds. Shortly after landing in Hamburg I was awestruck by the snow and the distinct holiday spirit, but even that did nothing to prepare me for arriving at the actual venue and seeing for myself what CCC was.

The convention center in Hamburg is somewhat tricky to navigate on a usual day. When CCC took over the place, the entire building would be cast into darkness, and as you'd wander around lost you'd be assaulted on all sides by colorful blinking lights and rugged, individual technical enterprise. On your left, someone's built a human-size copy of Tetris out of cubic LEDs. On your right, someone's running a lockpicking workshop. Above you a 3D- printed drone flies around in a circle while the person who printed the thing considers how much effort would be required to make it automatically mix milkshakes and deliver them to random conference participants. You crash on a sofa and to your left is some person who hasn't slept in 50 hours, who's had their third bottle of Club Mate, their caffeine-powered fingers typing furiously at their laptop keyboard and producing a gigantic blob of hexadecimal output. Above the two of you hangs a large handmade banner that proclaims, "Be Excellent to Each Other!".



# THE PROBLEMS THEY DEAL WITH ARE OF THE MOST DIFFICULT AND ADVERSARIAL FOUND ANYWHERE IN INFORMATION SECURITY

I spent the first day trying to take in all of that. Later, when I desperately tried to describe to my wife where I was and what I had seen, the best I could do was compare it to Harry Potter's culture shock when he visits Diagon Alley for the first time and finally sees first-hand all these wizards practicing their wizardry (the usual biting retort to such comparisons goes, "Read Another Book"). She said "ok, cool", and I felt that somehow a lot of the experience had been lost in transmission. But anyone who has actually been to CCC will understand.

Since then, I've spent a long decade getting to know this world that I first visited in Hamburg. I've heard plenty of amazing and unlikely research shared by the Cryptographers, the Reverse Engineers, the Threat Intelligence people and the Hardware Hackers—but the most alluring and difficult part has been understanding the world of vulnerability researchers. The problems they deal with are of the most difficult and adversarial found anywhere in information security; their work is probably the most emblematic of the field, and the likeliest to reach the average person via the evening news.

I wish I'd had someone explain it all to me right then and there, and hopefully the next person in my shoes will run across this article and find it to suit their need. But before we jump into all the talk about one-days and cross-site reference forgeries and whatnot, I first wanted to bring up CCC, and that banner that says "be excellent to each other". They may not be the most technical or edgy aspect of information security or vulnerability research, but they're why I care enough to have written this.

## Our Preemptive Apology

This article mentions by name many researchers, vulnerabilities, terms, concepts and legal entities. If we have misunderstood some piece of history, misattributed an achievement, failed to give due credit, incorrectly characterized a person or a company—our deep apologies, and please contact us so we can fix what needs fixing.

# What is Vulnerability Research?

There's a different answer in theory and in practice.

In theory, vulnerability research is the capital-A Art of understanding systems so thoroughly that it becomes possible to craft unexpected input that makes them behave in unexpected, typically disastrous, ways. Our own "A First Introduction to Systems Exploitation", mentioned above, has the following to say:

That part of information security that your parents warned you about [...] Systems are understood in terms of naked primitives; convenient abstractions are stripped away, or are unavailable to begin with. The narrative about how the system is "supposed to" behave is ignored with prejudice. These systems are then understood in more detail than before, and may even be made to behave in ways that they shouldn't.

The eminent [Jargon File](#) puts it [more bluntly](#):

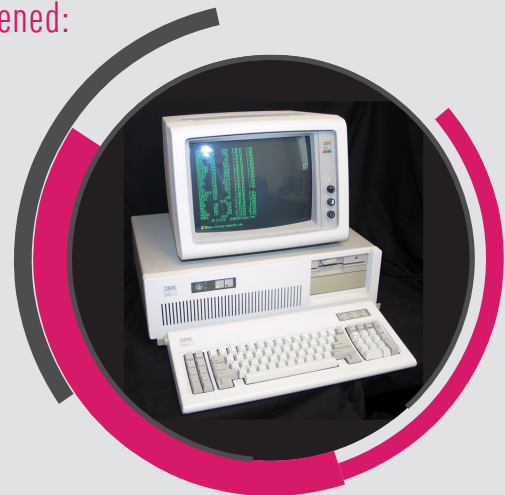
"The word hack doesn't really have 69 different meanings", according to MIT hacker Phil Agre. "In fact, hack has only one meaning, an extremely subtle and profound one which defies articulation. Which connotation is implied by a given use of the word depends in similarly profound ways on the context [...] Hacking might be characterized as 'an appropriate application of ingenuity'. Whether the result is a quick-and-dirty patchwork job or a carefully crafted work of art, you have to admire the cleverness that went into it."

But also provides [the following story](#), which we really feel is the quintessential definition of "hack" by example:

[..] One day an MIT hacker was in a motorcycle accident and broke his leg. He had to stay in the hospital quite a while, and got restless because he couldn't hack. Two of his friends therefore took a terminal and a modem for it to the hospital, so that he could use the computer by telephone from his hospital bed. Now this happened some years before the spread of home computers, and computer terminals were not a familiar sight to the average person. When the two friends got to the hospital, a guard stopped them and asked what they were carrying. They explained that they wanted to take a computer terminal to their friend who was a patient.

The guard got out his list of things that patients were permitted to have in their rooms: TV, radio, electric razor, typewriter, tape player, ... no computer terminals. Computer terminals weren't on the list, so the guard wouldn't let it in. Rules are rules, you know. [..] Fair enough, said the two friends, and they left again. They were frustrated, of course, because they knew that the terminal was as harmless as a TV or anything else on the list... which gave them an idea.

The next day they returned, and the same thing happened: a guard stopped them and asked what they were carrying. They said: "This is a TV typewriter!" The guard was skeptical, so they plugged it in and demonstrated it. "See? You just type on the keyboard and what you type shows up on the TV screen." Now the guard didn't stop to think about how utterly useless a typewriter would be that didn't produce any paper copies of what you typed; but this was clearly a TV typewriter, no doubt about it. So he checked his list: "A TV is all right, a typewriter is all right ... okay, take it on in!"





So, that's theory. Practice is, predictably, grittier. The discovery of a high-impact vulnerability can equal money, power, prestige or the satisfaction of averting future disaster. Hence, for every undiscovered such vulnerability, there is an implicit wacky race of highly motivated freelancers, nation-state actors, security researchers at big tech, graduate students and other players, each armed with their respective tools, expertise and preponderance of free time, all vying to be first past the post. For these actors, there are few things more satisfying than finding a vulnerability with wide reach and crippling impact (in these modern days, one might add: wide enough reach and crippling enough impact to warrant [a catchy name, a dedicated web page and an imposing logo](#)). This diverse supply of vulnerabilities meets a diverse demand: actors who have actual plans of what to do with a vulnerability, including some (not all) of the actors mentioned above, will happily pay for one handsomely in lieu of doing the difficult research.

Nation-state actors of course qualify here; ordinary cybercriminals mostly don't, due to pure cost-benefit considerations. After all, a vulnerability unknown to the world at large—what's known as a "zero-day", a term that originated at

"Warez" bulletin boards during the 1990s—might net a godly initial conversion rate of victims; but such vulnerabilities [can easily go for hundreds of thousands of dollars](#), a sum comparable to [the total yield of a highly successful ransomware campaign spanning several months](#) and even [the total monetary yield](#) of the infamous [2017 Wannacry outbreak](#). Consider: even that latter incident, widely seen as the perfect storm and the contemporary upper bound for the scale of commodity malware, could not break the six-figure barrier. It's very difficult to imagine a visionary cybercriminal so ambitious and so confident, planning a cyber-heist on such an unprecedented scale, that would take the risk and eat the upfront investment to power their campaign with a zero-day vulnerability. Much easier to stick to malicious spam, which is [reliable and affordable even if you're some broke nobody in Nigeria](#). Even Wannacry was able to wreak all the havoc it did by exploiting a mere "one-day" (an already widely-known vulnerability) in Microsoft SMB, a patch for which had been available for a full two months before the attack. If you're targeting indiscriminately, why pay an exorbitant amount for artisanal secret research when you can just play a numbers' game and count on enough victims to click "enable macros" or not click "update and restart"?

While a zero-day vulnerability can fetch a pretty penny being sold to whatever interested party, there are arguably ethical issues with that. Suppose Alice finds a vulnerability, and just goes to the dark web and sells this new-found weapon to the highest bidder, Bob, without expressing any interest in what kind of enemies Bob has made that put him in need of such a weapon, and what he intends to do to these enemies once he properly takes aim with it. If Bob is a dictatorial despot in some seventh-world country, and his intended use for Alice's discovery is to track, capture and torture dissidents and pesky journalists, some would argue that Alice does not come out of this as quite the paragon of virtue. If that bothers her, she can instead disclose her discovery to the owner of the vulnerable technology, through proper channels, following rigorous protocols designed to ensure that by the time the vulnerability is made public, a working patch that fixes the vulnerability in the offending technology is already available. Crucially, companies (more typically tech giants) have recognized that Alice has perverse incentives




here, and that to serve the greater good she must forego an obscene amount of money; recognizing that many among us aren't quite that saintly, these companies have set up "Bug Bounty" programs that try to offer competitive compensation for coordinated disclosure of vulnerabilities. For instance, Microsoft [offers a bounty of up to \\$250,000 for "critical remote code execution, information disclosure and denial of services vulnerabilities in Hyper-V"](#), its hypervisor product.

There is also a "gray market" for vulnerabilities: some actors with visible websites, physical addresses, real names, and other such signifiers of above-board conduct will pay Alice for her discovery conditional on no public disclosure, and her trust in their judgement of who to sell this information on to and when. One such actor is [Zerodium](#), founded in 2015 by the members of a now-defunct French infosec firm called [Vupen Security](#), which specialized in discovering vulnerabilities and selling them to law enforcement and intelligence agencies. The client pitch at the Zerodium website reads, "access to [our] solutions and capabilities is highly restricted and is only available to a very limited number of eligible organizations".

# MITRE IS A US-BASED NONPROFIT THAT DATES BACK ALL THE WAY TO 1958

**Severity** CVSS Version 3.x CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**



**NIST:** NVD

**Base Score:** 8.1 HIGH

**Vector:** CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

*NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA.*

*Note: NVD Analysts have published a CVSS score for this CVE based on publicly available information at the time of analysis. The CNA has not provided a score within the CVE List.*

## Severity of a Vulnerability

[MITRE](#) is a US-based nonprofit that dates back all the way to 1958 (“MITRE” does not stand for anything). Initially involved with applications of early computing technology to military and civil engineering projects, MITRE soon expanded to develop advanced communication and early warning systems (there’s also that [research paper about natural eradication of cannabis in the western sphere of influence via biological warfare](#); not very representative of MITRE’s general work, but we can’t not mention this). In 1999, MITRE launched the Common Vulnerabilities and Exposures (CVE) glossary—a database that records publicly-known vulnerabilities, and has become a de-facto canonical reference. The vulnerability exploited in

2017 by the aforementioned Wannacry malware to create a rampant cyber pandemic and cause untold amount of damage is catalogued there by the dry, detached designation “[CVE-2017-0144](#)”, ala [The SCP Foundation](#). The database has recorded 12,174 new vulnerabilities in 2019 alone (For comparison, the early 2000s saw about 1,500 new vulnerabilities per year, and the early 2010s—about 5,000). Vulnerabilities are ranked for severity by the 0-to-10 Common Vulnerability Scoring System (CVSS), which assigns each vulnerability a score based on various properties. The current version of the scale, CVSS 3.1, computes a score based on the following questions:



Q

How feasible is the **attack vector**? Can you launch the attack from halfway across the earth, or do you need to be in the same LAN as the victim? Or do you need to be literally running code on the victim machine, or even to have physical access?

Q

How **complex** is carrying out the attack in practice? Can you just run a piece of code and expect immediate success, “script kiddie” style, or would you need to study your target carefully, make individual preparations and hope you get lucky?

Q

What **privileges** are required up-front? Can you launch the attack as an anonymous nobody, or do you require at least a working user account or some such? Or do you need to have administrator privileges, which are even further abused beyond what the system’s design intended?

Q

Is **user interaction** required? Does the victim have to click “ok” on some prompt or do they get, as the phrase goes, “pwned”, without even that opportunity?

Q

Can the **scope** of the impact grow to include systems that weren’t directly attacked? For example, a compromised SQL server that can be trivially used to compromise other SQL servers, or a compromised website that can be trivially used to execute malicious scripts on the browsers of unsuspecting visitors.

Q

How impacted are the **Confidentiality, Integrity** and **Availability** of the targeted system? Can the attacker tamper with information, learn secrets, and/or disrupt services? For each such ability, is it constrained or does the attacker have free reign?

Q

How about a **working exploit**? Is it even proven to exist? If so, is it just a Proof of Concept that allegedly worked once in someone’s lab, or is there a mature exploit that will work most of the time—or even a complete proliferation of a reliable and easy to use automated exploit tool?

Q

How thoroughly has the vulnerability been **remediated**? Is there an official patch that resolves the issue? Maybe just a temporary band-aid, issued by the targeted technology’s vendor to be used while the official patch is being worked on? Or maybe even just an unofficial workaround that those in the know can apply? Or, in the worst case scenario, no remediation at all?

Q

Are there any **special circumstances** that should change our assessment, beyond these dry details? e.g. if the component under question is compromised, can this easily lead to a proper catastrophe, casualties, social unrest or other cruel and unusual damage?

Q

How **confident** are we that the vulnerability exists at all? Do we know its root cause and the mechanism behind it? Are we fairly certain that the vulnerable behavior can be reproduced? Has the relevant vendor confirmed it?

Q

To what degree does the compromised component have the ability to *propagate further into the environment*?

Some of these questions are intrinsic to the vulnerability, and the answers to them will remain the same throughout its lifetime; others will evolve as time passes. The answers can be represented as a terse string called a CVSS vector and aggregated into an overall score, based on a well-defined standard numerical scale. For instance, the aforementioned CVE-2017-0144 has the vector CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H, and an overall score of 8.1 out of 10 ([this calculator](#) delves in-depth to how these map to answers to the questions above).

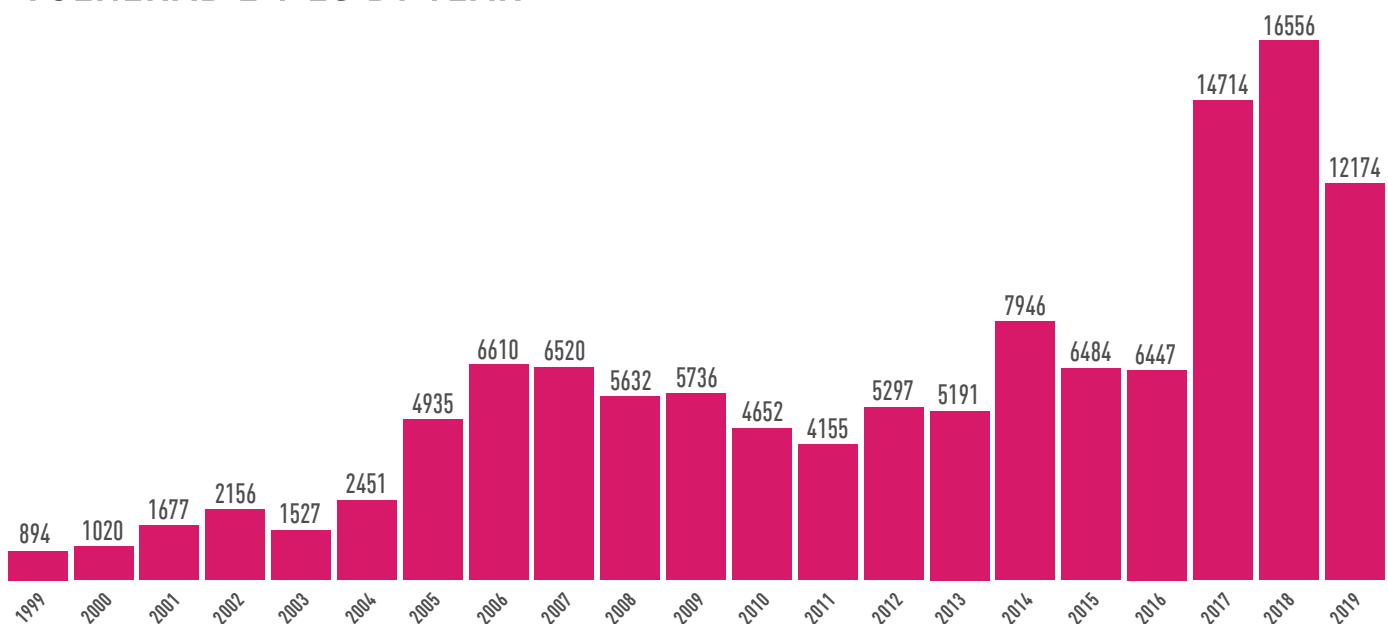
## Lifecycle of a Vulnerability

How many critical vulnerabilities are out there right now, as we speak, waiting to be discovered? Given the growing rate at which new ones are found, some of which astoundingly simple in concept and ease of exploitation (see e.g. ShellShock below), it is

easy to imagine a seething, unknown underworld of undiscovered vulnerabilities—a mass of dark matter that we can only have a fleeting experience of when a piece of it comes to light. Chances are, there are many possible sequences of bytes that could each compromise every Windows 10 machine in existence, and the world can only operate in a sane way by virtue of bliss ignorance. Of most vulnerabilities, no one knows and no one will ever know.

Out of those many, many possible vulnerabilities, one will—against all odds—be teased out by the investigative glare of a researcher (we're sure there's a tortured metaphor about the beginning of life somewhere in there). A researcher sets their sights on an application to be analyzed because it's popular, or has a particularly high impact if compromised, or they suspect the code's security standard is not quite up to par, or just because it seems interesting to the researcher personally. They observe the application's source code—or, if they lack access to it, the raw assembly.

## VULNERABILITIES BY YEAR



# ONCE A BUG IS FOUND AND UNDERSTOOD, THERE IS SOMETIMES STILL A WAY TO GO UNTIL A FULLY-FLEDGED VULNERABILITY

The old-school way to proceed is to look for places where the program is exposed to input, manually understand the code that processes this input and then reason about the way that this code might malfunction. In recent years, the paradigm of “fuzzing” saw a great increase in popularity—automated tools, such as [AFL](#), will spam the targeted application with a great amount of pseudo-random input, note when some input causes the program to behave in a different or interesting way, and then use that input as a starting point when generating even more input, basically using a genetic algorithm that is reaching for some truly pathological edge case that no human would have thought of, hopefully crashing the program and indicating a bug that can be looked into further.

With the concrete lead of a crash in hand, the researcher can then proceed to reason about the code at fault and understand the root cause of the crash (the answer is typically one of the items in the section below, “Causes and Effects”). To properly appreciate the strength of fuzzing as a paradigm, we recommend reading the article [50 CVEs in 50 Days](#), where the authors “took one of the most common Windows fuzzing frameworks, WinAFL, and aimed it at Adobe Reader, which is one of the most popular software products in the world [...]” and “[found] over 50 new vulnerabilities in Adobe Reader [...] [which is] 1 vulnerability per day—not quite the usual pace for this kind of research.”

Once a bug is found and understood, there is sometimes still a way to go until a fully-fledged vulnerability, and a way from there to a working exploit. The researcher must understand the degree to which they control the terms on which the application malfunctions, and how these can be used to manipulate it. These considerations are usually straightforward when dealing with injections, request forgeries and other types of vulnerabilities some levels of abstraction above raw assembly; but when dealing with the raw assembly, a researcher will typically have to answer difficult questions such as “what can I write? Where? What constraints are there on my input—do I have to avoid null bytes, or even non-printable characters altogether? Is there somewhere I can write some assembly, and make the program transfer control to it?”. This whole chain of questions is usually relevant when aiming for full arbitrary code execution, but even a lesser vulnerability will typically require a researcher to deal with at least some of these considerations. Even when they do, their work is not done: as a rule, applications are expecting to be attacked, and mitigations will be in place. “Canaries” may be placed at the end of buffers, inducing programs to screech and die the moment they are overwritten, rather than let an attacker get away with a successful buffer overflow (more about this below). Pieces of memory may be marked as “non-executable” by the operating system, preventing the attacker from a simple “write code, then jump to it” (this mitigation is

called Executable-space Protection, though the Windows-specific term, Data Execution Prevention (DEP), is better known). This forces them to creatively use machine code that’s already in place and meant to be executed (the most widely-known method of doing this is called ROP—Return Oriented Programming), but then the OS, wise to this trick, randomizes the position of process code in memory, forcing the attacker to figure that unknown out before they can even get to their attack proper. Once the attack is executed, instead of the promised land of endless privileges, the attacker might find themselves in a Sandbox—a restricted environment with few privileges, or even a Docker container or a Virtual Machine. They will have to “escape” this environment before they can proceed. Each way of getting around a mitigation they find leads to a new mitigation, put in place to frustrate them; every new such mitigation leads to more clever, elaborate and unintuitive workarounds. It’s the sort of thing the phrase “cat-and-mouse game” was invented for. Eventually, the attacker may despair of their lead as non-exploitable, but of course they root all the

while for the second possibility—that they triumph, and emerge from their trials and tribulations with a working exploit in hand.

When an exploitable vulnerability is first discovered by a researcher, it is considered a “Zero Day”—meaning there is no available patch for it, and knowledge of it is limited to its discoverers and possibly a small circle of other parties (again, this term originated at “WareZ” bulletin boards during the 1990s, and referred originally to brand new software, off the shelves of non-consenting developers, available for download hours after its official release). Once a patch is widely available, the vulnerability becomes a “One Day”, and use of it for exploitation hinges on whether the attacker can strike before the defender applies the patch. This is why Windows nags you so unrelentingly to restart your machine and install your updates. Apart from Zero-days and One-days, periodically someone will try to assign meaning to a similar term using some other number than Zero and One, with varying degrees of success and intended irony.

```

american fuzzy lop ++2.59d (loadtiff) [explore] (0)
┌───────────┴───────────┐
└ process timing        │ overall results
   run time  : 0 days, 8 hrs, 14 min, 54 sec │ cycles done : 0
   last new path : 0 days, 0 hrs, 12 min, 28 sec │ total paths : 525
   last uniq crash : none seen yet           │ uniq crashes : 0
   last uniq hang  : none seen yet           │  uniq hangs  : 0
└── cycle progress ───┘ └── map coverage ───┘

```

Configuring update for Windows 10  
5% complete  
Do not turn off your computer

Wikipedia off-handedly mentions a concept of [half-day vulnerabilities](#), where “few users are aware and implementing [the] patches”. A security vendor once famously touted its “Negative-day protection”, which provoked sardonic tweets wondering if the technology in question will mitigate a vulnerability by automatically identifying the researcher who discovered it and traveling back in time to assassinate their grandfather.

Assuming the researcher in question escapes such a fate, we’ve already discussed their options: keeping the vulnerability to themselves, selling it to an interested actor, or disclosing it to the affected software vendor. If they pick one of the first two options, this is where vulnerability research ends and exploitation begins. Money may change hands and unwitting targets may be compromised under cover of darkness, until the vulnerability comes to light and is patched—if and when that happens. The third option is what’s commonly called “coordinated disclosure” and, effectively, constitutes an entire additional phase of the research. It is an involved, protracted and sometimes exhausting legal tango, with its prize typically being a monetary “bounty” paid by the affected vendor, and the researcher having bragging rights and being secure in the knowledge that they did The Right Thing™.

To be blunt, the above is putting an idealized gloss over what is sometimes a nasty, adversarial situation. Not all vendors see the value in incentivizing researchers to poke around products and look for vulnerabilities. Some will fix the vulnerability, but will not offer monetary compensation; some will shrug and refuse to fix the vulnerability—effectively playing a game of “chicken” with the researcher, the implicit threat being “oh, you care so much about how this will affect people? Go ahead, let’s see you disclose it when we haven’t released a patch”. Finally, some will even resort to legal threats—one vendor’s CSO famously went on a Twitter rant on this subject in 2015, scolding the community thus:

Customers Should Not and Must Not reverse engineer our code [...] it’s our job to do that, we are pretty good at it. [...] Many companies are screaming, fainting, and throwing underwear at security researchers to find problems in their code and insisting that This Is The Way, Walk In It [...] You can’t really expect us to say ‘thank you for breaking the license agreement.’

This tweet was later removed, but the sentiment behind it lingers in the industry—so much so that the [“pwnie” awards](#), hosted annually at the Black Hat conference, have a “Lamest Vendor Response” category just for this sort of thing.

Let’s be optimistic, though, and assume that the researcher has a cooperative dialogue partner in the affected vendor (even if their cooperation does not extend all the way to screaming and underwear-throwing). The vendor will require time to fully understand the vulnerability, and then issue a response. The researcher is hoping for an enthusiastic “oh god, we have to fix this immediately”, but they aren’t always so lucky. Sometimes vendors will require a fully working exploit before taking a vulnerability seriously, sometimes they will dismiss the vulnerability as “out of scope” for their bug-hunting efforts, and—as mentioned above—sometimes they will not respond at all. This latter gambit used to be so popular that eventually a standard emerged (often credited to Google’s Project Zero; more on them below) where researchers declare a generous time window for the vendor to fix the vulnerability (a reasonable figure is 90 days), after which they will regretfully go public with their findings, patch or no patch. This might cause damage in the short run, but in the long run, it is the only deterrent against a future where vulnerabilities accumulate without end and vendors sweep them under the rug.

Hopefully, this process does end with the vendor publishing a patch. At this point, the researcher and vendor typically co-publish a disclosure of the nature of the vulnerability, affected product version, and the patch. The vulnerability turns into a “One Day”, and so begins the rush of malicious actors to exploit the vulnerability before victims manage to patch their software. Some vendors enjoy the privileged position of being able to patch customers’ software remotely and discreetly; for example, such is the case with Microsoft. Once a month, they release a bundle of patches that are silently installed on every network-connected Windows Machine with a functioning update mechanism. Since this typically happens on the second Tuesday of every month, it is colloquially known as “patch Tuesday”. In these cases, the ability of malicious actors to abuse One Days is mitigated somewhat.

ONCE A MONTH, THEY RELEASE A BUNDLE  
OF PATCHES THAT ARE SILENTLY INSTALLED ON  
EVERY NETWORK-CONNECTED WINDOWS MACHINE  
WITH A FUNCTIONING UPDATE MECHANISM.

## Causes and Effects

All the above is good and fine, but how does a vulnerability *work*? Some input is crafted that makes a music player or a website gag, writhe and behave in a way its developer never anticipated—how can such a thing happen? While this is not a technical document, we would be remiss not to introduce you to the terminology, and what it means, especially since often the underlying principle is simpler than it seems at first. This is by no means an exhaustive list, but it includes many of the recurring themes in vulnerability root causes and impacts. We will try to explain each term shortly and succinctly.

---

### Causes: How an *exploit* becomes possible

#### Buffer Overflow

In French, this attack is called [Dépassement de Tampon](#). We just felt like we should open with that.

A process often needs a chunk of memory, a “buffer”, for some purpose, and later writes to it. Suppose an attacker can control the data to be written and the length of the data is not verified, or verified incorrectly; the attacker can then craft data longer than the buffer length that when written to the buffer, will overwrite process memory that *follows* the buffer, which the attacker was not meant to access originally. e.g. Alice is very thirsty and asks Mallory “do you have the time?”. In a moment she will be thinking to herself, Mallory just said the time is \_\_:\_\_. I’m very thirsty, with Mallory’s answer meant to overwrite the blank. Mallory responds, 12:47. I am not. This overwrites the blank and some characters that follow it. Now Alice is thinking Mallory just said that time is 12:47. I am not thirsty, and soon dies of dehydration.

---

#### Integer Overflow

A process keeps track of some quantity, which an attacker can add to. The attacker makes the quantity grow and grow, until finally the underlying memory literally runs out of ways to represent a larger number. The CPU carries out the addition anyway, because *in certain situations*, this is useful to do and produces a result that makes sense. Not this specific situation, though. Now the process is convinced that some shopping cart contains a negative two billion apples.

## Use After Free (UAF) and Misc Memory Management

A process is responsible for keeping track of what memory it is using, where, and what for. It may mistakenly declare itself done with some memory, then a moment later try to use that memory as if there's something useful still there. An attacker might have intervened in the meantime, requested the unused memory and written into it, with unexpected results. e.g. Alice throws away her old grocery list. A week later, she makes a mistake and forgets about this, and goes looking for the old grocery list again in order to have it scanned by her online shopping app. She finds the list in the garbage bin outside, where Mallory had found it a day before and added two tons of creamed corn to it, to be delivered to Alice's doorstep. Alice has her app scan the list, and is soon set back some \$10,000 and a hefty fine for domestic disturbance.

UAF is one kind out of a larger class of such "memory management" errors: A process can also declare itself done



SHAKESPEARE QUOTE OF THE DAY

An SSL error has occurred and a secure connection to the server cannot be made.

with memory that it had declared itself done with *already* (Double Free), or try to ask for some memory, get the response "memory allocation failed" and then say "great! Please write this-and-that content to 'Memory Allocation Failed'" (Null Dereference). This last one has had such destructive consequences over the years that Computer Scientist Tony Hoare, who had worked on the ALGOL programming language during the 1960s, outright apologized in 2009 for bringing null references into the world:

---

## Type Confusion

A process is responsible for interpreting bits in memory—does this sequence of bits represent a number, a list, an internet connection? If the process can be tricked into treating a blob of bits as one thing one moment, and another thing the next, funny things can happen. e.g. Alice is very scatter-brained. Mallory calls her and says "hey, can you please open that list you've been keeping on your phone of people and the number of times they've said the word 'Longitude' – the one that says 'phone book' on the icon. I just heard your husband say it today." Alice duly increments the number by 1, and the next time her husband calls, she innocently asks "hi, who's this". This is the last straw for an already strained relationship, and it ultimately snowballs into a long, ugly divorce.



## Injection

A process is responsible for keeping its own internal logic separate from user input. If this separation fails, an attacker can craft input that gets evaluated as if it were process internal logic. An easy, extreme example of this is a ‘greet’ script that asks for the user’s name, transplants it into the command `print("Hi, {name}!")` and executes the resulting command. An attacker can answer that their name is `Bob"); delete _hard_drive(); //` (where `//` means “ignore rest of line”). The server will execute the command `print("Hi, Bob"); delete _hard_drive(); //!`, certainly not functionality originally intended by the script’s original author. A related concept is a “path traversal” attack, where e.g. Mallory gets to pick a file name for saving a new picture in `/home/Mallory/Pictures`, and she picks `../../../../important_scripts/script.sh` and thus gets to overwrite its contents, which clearly no one intended. Another related concept is the Cross-Site Scripting (XSS) attack; the simplest variant of it basically involves some innocent `printname.html` webpage that executes `print("Hi, {name}!")` and gets blindsided by a smartalec attacker who’s wandering the web and distributing links to `printname.html?name='Bob"); delete _hard_drive(); //'`. To mitigate this issue, many applications implement “input sanitization” and reject suspicious input, or create a more delineated divide between pre-compiled main logic and the input that it processes (if you’ve ever wondered why Python’s `subprocess` module won’t let you call `call("echo hello")`, but insists on either the safer, idiomatic `call(["echo", "hello"])` or the “we hope you know what you’re doing” override, `call("echo hello", shell=True)`—this is the reason).

---

## Cache Poisoning

Some processes act as servers; clients send requests their way, and they respond. Many of the requests repeat, and so these servers keep a cache of recent Frequently Asked Questions. If these servers compute answers completely on their own, without dependence on outside input, that’s that; but if they do depend on outside input in some way, and are too trusting of it, an attacker can manipulate this input at just the right time, changing the “official answer” in the FAQ for hours, days or weeks until the server thinks to do another reality check. A famous attack of this type was what’s called “ARP poisoning”, where an attacker would spam the local network with declarations that their machine is the network’s default gateway, causing all traffic to be routed through them. This specific attack carries much less weight today, in a world that is wary of it and has gone extremely wireless. *Someone* can probably listen in on your traffic, anyway, and not using an encrypted connection is unjustifiable negligence (ask the [Wall of Sheep](#) folk about this). But the more general principle of cache poisoning is still relevant.

## Flooding

Sometimes a system can withstand an attack, but not a million simultaneous copies of that same attack. Suppose you're in charge of an internet service and one day, without warning, tens of millions of different machines across the globe begin [bombarding your servers with requests, the aggregate size total of which reaches 10 Wikipedias per second](#). What do your servers do? Choke and die, that's what. While flooding is most synonymous with this sort of "denial of service" scenario, and in that capacity depends more on an attacker's control of many machines than on their access to esoteric domain knowledge of the attacked system, one can make a case that e.g. the exploit for CVE-2008-1447, explained below, involves a form of flooding (and, of course, be promptly clubbed to death by linguistic prescriptivists).

---

## Replay Attack

Sometimes a server should only perform some action after validating that the request comes from a client with the proper credentials. If the protocol used for validation is naive enough, someone listening in—or even someone with access to artifacts left over after the interaction—can use what they've seen to impersonate someone who has the proper credentials. The simplest example of this is the timeless scenario, found in many a video game, of some location entrance blocked by a guard who demands to hear a password; the player simply lurks around the scene for a while, and invariably someone comes along and produces the password, which the player can then repeat to gain entry. These attacks are typically mitigated by challenge- response schemes, randomization and explicitly tying challenges to the responder's identity. (One can imagine an 'Infosec Speakeasy' skit, where the owner grabs a prospective customer and growls, "Stop right there! What's `sha256(the password || your name || the current timestamp)?`"). The "Pass the Hash" attack against the NTLM Security protocol relies on a similar principle; an attacker can authenticate as a user without access to their sensitive and well-protected password, and instead "replay" a derived protocol artifact (the hashed password) that is kept in store and was never meant to be used this way. One can also say something similar about Session Hijacking and "Pass the Cookie" attacks that steal and re- use the victim's web browser cookies, but whereas the NTLM bug was just a mistake, browser cookies work that way by design and are admittedly very convenient.

## Race Condition

Computation that happens in parallel (or accidentally in parallel!) is infamous for often violating programmer assumptions about what happens first and what happens later, and therefore being particularly difficult to debug and reason about (the joke goes: “some people, when confronted with a problem, think ‘I know, I’ll use multithreading’.”) But, whether we like it or not, we live in a parallel world; a programmer cannot even assume that two adjacent assembly instructions generated by their code will run one directly after the other, without meaningful events happening in-between, unless they arrange for this guarantee carefully and explicitly. A classic flavor of race condition is the Time of Check to Time of Use (TOCTOU) vulnerability, where the system makes an access control decision based on a prompt that completely changes its meaning by the time the decision is implemented. e.g. A server might naively execute the following logic: `does Mallory own this file? If so, then ok, open this file and show her its contents; Mallory arranges for the file to be a shortcut to one of her personal documents, and once the then is reached, quickly changes the shortcut to point to the server’s master password database instead while the shortcut name remains the same.`

## Request Forgery

A process should use its privileges responsibly and not just blindly pass on every request made to it by a less privileged entity (be it a user, a document, or anything else). If the process does not apply a sanity check of “does it make sense for this entity to ask me to do this thing”, its negligence can lead to an attacker tricking the process into applying its authority in the attacker’s service. This is also called a “confused deputy” attack, and its classic flavor is what’s called a Cross-Site Request Forgery (CSRF) attack—where e.g. Mallory’s personal webpage contains a surreptitious request for the resource `https://restaurant-review.com/mallorys_place/review.html?star_rating=5`;

if a web browser blindly agrees to make this request, anyone using it while logged into `restaurant-review.com` who then visits Mallory’s website will be made to rate her restaurant 5 stars without even knowing this happened. Nowadays, this attack is mitigated by a feature called “same-origin policy”, which does not allow scripts running on a web page to read information sent by a web page belonging to a different domain; so if `restaurant-review.com` requires any meaningful interaction, and it typically will, Mallory’s plot is foiled. When this general method is used to compromise servers, rather than clients, it’s called Server-Side Request Forgery (SSRF); one prominent flavor of it is the XML External Entity attack, where the server is fed a request referring to some XML resource and is made to evaluate it blindly (similarly to how the web browser blindly evaluates the crafty request in Mallory’s website).



## Semantic Bug

We would be in dereliction of our duty not to mention that some bugs are domain-specific, inherent to design rather than implementation, cool and unusual. They don't fall into any of the specific categories above, and arise out of the very specific circumstances of what the application in question is trying to do, and some *subtle* way in which it is doing almost that thing, but *not quite*. Kaminsky's DNS bug falls under this category, as do Spectre, Meltdown, Curveball and undoubtedly many other vulnerabilities with less pushy PR. Thanks to these, we have the joy of reading through a long list of terse vulnerability descriptions: "buffer overflow... integer underflow... SQL injection... buffer overflow... buffer overflow..." and then suddenly, "induced mismatch between actual and interpolated pathfinding during initial learning phase, leading to arbitrary duplication of coupons". Don't tell anyone, but these are our favorites.

---

## Effects: *What* impact a successful exploit has on the system

### Privilege Escalation

An attacker acquires privileges that they did not have before, without being properly vetted by the system (or by being "vetted" so easily that the original application design obviously could have not and should have not intended for this to happen). Request forgeries typically achieve a form of privilege escalation; for instance, the attacker starts with mere access to the content displayed in a web page, and is able to leverage this and induce the much more capable web browser to act on their behalf.

---

### Information Disclosure

An entity acquires information that it should otherwise not be able to access. For example, in the scenario described above for a "Race Condition" vulnerability, Mallory is able to access the list of all user passwords, which she should not be able to do. Heartbleed (CVE-2014-0160), expanded upon below, is a classic example of an information disclosure vulnerability. An attacker exploiting it cannot run code on the victim machine, but they can learn a great deal.

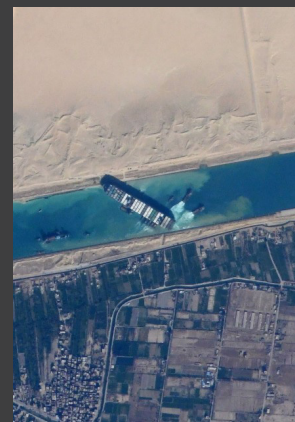
## Arbitrary Code Execution

Probably the most severe possible consequence; the attacker owns the victim application now and can have their way with it. This is different from what is achieved by a command injection, where an attacker can merely run (e.g.) their choice of SQL; it is the semantic equivalent of being able to log into the server and run an executable file of your choice, with the same privileges as the compromised application. At this point, the only thing preventing the attacker from outright owning the victim machine is the possibly limited privileges of the compromised application. If that application happens to have administrator privileges, that makes the attacker the machine's new administrator.

---

## Denial of Service

The attacker is able to knock a certain service offline and render it unusable. All other things being equal, this attack sits relatively low on the totem pole with respect to possible impact and required sophistication—as mentioned above, in the typical scenario, no esoteric domain knowledge of the targeted system is required, no information is leaked and attackers do not have their way with the system, except insofar they force it to stop functioning. But in a pathologically interconnected world where every functioning company, institution and piece of infrastructure depends on 270 others, and one ship stuck in a canal can cause monetary damage on par with the GDP of a small country, a little denial of service can go a long way.



## Vulnerabilities of Note

We earlier talked about “motivated freelancers, nation-state actors, security researchers at big tech, graduate students and other players, each armed with their respective tools, expertise and preponderance of free time”, searching for vulnerabilities and hoping to hit the next big one. We may be able to get a more concrete grasp of what this means if we look at some of the most high-profile vulnerabilities of recent years—their causes, and some of the involved names and faces.

### [CVE-2019-6111](#)

An improper input validation vulnerability in OpenSSH’s SCP client, which is used for file transfer over SSH protocol. The implementation of SCP was based on an ancient unix program called rcp, which dates back to 1983; in the protocol used by rcp, the server gets to specify which files and directories are sent to the client. So broadly speaking, all that’s standing between a malicious server and free reign to overwrite the victim file system is the client’s vigilance when validating the server response—which turned out to be insufficient. This vulnerability was [discovered](#) by Harry Sintonen, senior security consultant at security vendor F- Secure. Sintonen has an [adorable minimalist personal website](#) which lists other vulnerabilities he’s found, such as [this header injection in the D-Link DGS-1250 network switch](#).

### [CVE-2019-0708 \(“Bluekeep”\)](#)

A use-after-free bug in Microsoft’s Remote Desktop Protocol (RDP) implementation. This vulnerability allows for remote code execution, and was considered to have potential to become a destructive “worm”—that is, spread from one vulnerable system across the network to new victims, and repeat. The discovery of this vulnerability was credited to the UK’s National Cyber Security Centre (NCSC), and its unusual nickname is due to Kevin Beaumont, who at the time was a security operations centre manager at [the Co-operative Group](#)—a British consumer co-operative with a diverse family of retail businesses including food, pharmaceuticals insurance services and funeralcare. Beaumont nicknamed the vulnerability “BlueKeep” because [“it’s about as secure as the Red Keep in Game of Thrones, and often leads to a blue screen of death when exploited”](#). Beaumont was [chastised tongue-in-cheek for “naming a patched vulnerability someone else found”](#), and he responded, “It’s so I don’t have to remember the CVE, I can’t handle numbers” (we can very definitely relate). As of 2020 he’s become a senior threat intelligence analyst at Microsoft and runs [DoublePulsar](#), a blog about “Cybersecurity from the Trenches”.

## [CVE-2020-0601 \(“Curveball”\)](#)

Well... imagine if entry into some high-security corporation required a reference from the CEO, but when giving your reference you were also allowed to supply the CEO’s phone number that’d be used to verify your reference with them personally, and on the way in no one bothered to cross-reference the number you supplied with the internal company records. This vulnerability is kind of like that, except the phone number is the parameters for some tough nut called the [discrete logarithm problem](#)—the difficulty of which your operating system ultimately relies on when verifying that webpages and applications were in fact authored by, say, Google, and not some guy in a Google suit. This vulnerability was discovered by an unsung hero at the NSA and then later disclosed (some information security folks sardonically noted that this may be the highest possible accolade for a vulnerability’s destructive potential—enough for the NSA to eventually say “we’re not keeping *that thing* a secret anymore”).

## [CVE-2020-1350 \(“SIGRed”\)](#)

An integer overflow bug in Microsoft DNS server that allows attackers to use malformed DNS response packets to run arbitrary code on the target machine. This vulnerability was the subject of a Department of Homeland Security emergency directive, instructing all government agencies to deploy patches or mitigations for it in 24 hours. It was [discovered](#) by Sagi Tzadik, a security researcher at Check Point. Scroll further down and you’ll find an interview with Sagi about his experience with SIGRed.

## [CVE-2018-8174 \(“Double Kill”\)](#)

A use-after-free bug in the VBScript engine used by Internet Explorer that allows arbitrary code execution by making IE chew on a maliciously crafted URL, which [for some perverse reason](#) can be done even if the victim just has IE on their system without ever using it themselves. This vulnerability was discovered under unfavorable circumstances, by researchers coming across active attacks in the wild already utilizing working exploits (these attacks have been attributed to a DPRK nation-state actor). This discovery was made independently by [Kaspersky](#) and [the Advanced Threat Response Team at 360 core security](#), who gave the vulnerability its name.

[Microsoft’s web page detailing this vulnerability](#) acknowledges by name Anton Ivanov and Vladislav Stolyarov from Kaspersky; Ivanov later became the Vice President of Threat Research at Kaspersky, still tweets now and then about strange new threats he’s run across (such as this [Piece of Linux Ransomware](#)), and recently [gave a presentation](#) at the [Security Analyst Summit](#) about the “WizardOpium” campaign, an attempt to target visitors of DPRK news sites via a zero- day vulnerability in Google Chrome.

## [CVE-2018-7600 \(“Drupalgeddon 2”\)](#)

A code injection vulnerability in Drupal, an open-source content management system used by over a million websites around the world including governments, retailers and financial institutions. When receiving certain input (FAPI AJAX requests, if you insist), a Drupal server would allow some of the input to “leak” into the server’s code itself, enabling an attacker to make the server execute code without administrator consent—this basically allows a remote hostile takeover of a website. Happily, this vulnerability was [discovered](#) by Jasper Mattsson, one of the contributors to the Drupal project. There was no need for an elaborate disclosure process; all that was left was to roll out the patch.

## [CVE-2017-11882](#)

A stack buffer overflow bug in MS-Office that allows a maliciously crafted document to run arbitrary code, making malicious documents a much more potent attack vector (as the victim, double click the document and you’re infected; you do not get a “please enable macros” prompt, do not pass “go” and do not collect \$200). Once publicly known, it became a favorite of commodity malware distributors as well as nation-state actors. This vulnerability, which at the time had existed undetected for 17 years(!), was discovered by security researchers at Embedi—a now-defunct cybersecurity startup company headquartered in Berkeley, USA, focused on immunizing IoT/embedded/smart end-point devices against 0- and 1-day attacks.

## [CVE-2017-5753 \(“Spectre”\) and CVE-2017-5754 \(“Meltdown”\)](#)

Well... imagine that your employer maintains this library full of documents, some classified and some not, and to read a classified document you have to present your credentials to the librarian. But the librarian is very short on time, so the moment you even express interest in any documents at all, she sends an errand boy to go fetch them from the back room just in case, even if the credentials will not check out; Also, the errand boy puts any requested documents into the librarian’s Quick Access Drawer under the front desk, because clearly these documents are popular now seeing as you just asked for them, and in five minutes some other person will probably be here to check them out again. So you turn to the librarian and say, “hey, go look at the company salary records for me, and if Bob’s salary is higher than mine, I want to borrow *War and Peace*; otherwise, I want to borrow *Romeo and Juliet*”. The errand boy heads out back, looks at the company salary records, finds out that Bob’s salary is *twice* yours and concludes that you’ll be borrowing *War and Peace*. On the way back, he discreetly puts *War and Peace* (and the salary records) into the quick access drawer, and reports to the librarian. The librarian adjusts her glasses, looks at you sternly and says, “excuse me, you’re not reading *War and Peace* or *Romeo or Juliet* or anything, your request started with ‘look into the company salary records...’ and you lack the proper credentials”. So you say “fine, I just want to borrow *War and Peace* then”. And the librarian is able to produce it *instantly* from her Quick Access Drawer, and thus answers your question about Bob without meaning to.



The Spectre vulnerability is kind of like that, except the librarian is the operating system, the errand boy is the CPU, the Quick Access Drawer is the cache and you're a rogue process with no permissions. This vulnerability broke all OS security boundaries and was at first dismissed by some in the semiconductor industry because it seemed too disastrous to be true. The discovery of this exotic vulnerability was a group effort that drew from people with many diverse backgrounds—including researchers from various academic institutions, all with a strong background in side-channel attacks, many of whom PhDs; co-author of SSL/TLS protocol who in a [candid personal bio](#) relates that he originally studied biology and planned to become a veterinarian; and a researcher from Google's Project Zero, a team of analysts tasked with hunting vulnerabilities (According to one xoogler, Project Zero was established [because](#) "It's a major source of frustration for people writing a secure product to depend on third party code [...] motivated attackers go for the weakest spot. It's all well and good to ride a motorcycle in a helmet, but it won't protect you if you're wearing a kimono.")

### [CVE-2015-0565](#)

A sandbox escape found in Google's Native Client sandbox, which is intended to allow running assembly inside a web browser. The attack was based on what's called a "rowhammer exploit", and is truly the stuff of black magic; even explaining it with a hand-wavey analogy would be a stretch. Let's just say that some types of RAM were built in such a way that enables an unintended effect—by reading certain bits repeatedly, an attacker can cause nearby bits to change their value, with disastrous potential results for security boundaries (a more involved technical explanation of the rowhammer effect appears [here](#)). The idea behind the attack was first proposed in 2014 in a joint paper by researchers from Carnegie Mellon university and Intel; a working implementation was delivered a year later in 2015 by Mark Seaborn, a member of the aforementioned Project Zero. He fed the sandbox legitimate-seeming instructions and then once these were approved by the sandbox, he abused rowhammer to flip bits and perturb the instructions into a slightly different form that could be used to escape the sandbox. Nothing about this is really particular to NaCl, and Seaborn [noted](#): "I picked NaCl as the first exploit target because I work on NaCl and have written proof-of-concept NaCl sandbox escapes before".

### [CVE-2014-6271 \("Shellshock"\)](#)

An injection vulnerability which leads to privilege escalation in bash, a popular command-line interface that's often installed as the default in unix-based systems. You might rightfully wonder how the phrase "vulnerability in bash" even parses—if you're supplying input to bash then you're running commands already, what does the vulnerability have left to do? As it turns out, upon startup bash would scan environment variables for the magic sequence `() {`, signifying a function exported as an environment variable; then once it encountered the matching `}` it would keep on evaluating and executing the included code, so that if you had an environment variable of the form `() {normal_func_definition}; execute_nasty_command`, every instance of bash would execute `execute_nasty_command` on startup. The crux is any Joe User could create these environment variables, which would then go on to be executed the next time the *system administrator* invokes bash, and thus by using shellshock every single user could effectively have system administrator privileges—a catastrophic result. This vulnerability was discovered by Stéphane Chazelas, UNIX/Linux and Telecom Specialist at SeeByte SeeByte Ltd.—a company specializing in software solutions for autonomous underwater vehicles.

## **CVE-2014-0160 (“Heartbleed”)**

A buffer overread vulnerability in the OpenSSL cryptography library. It allowed clients to read chunks of memory out of the server that they weren't supposed to, and every time we feel the need to explain it we just link to the [relevant xkcd](#), which explains the attack about as clearly as possible in 6 terse comic panels. The vulnerability was independently discovered by Neel Metha, an engineer at Google, who [related](#) how he found the bug following a “laborious” line-by-line review of OpenSSL's source code; and security firm Codenomicon, who provided the vulnerability's logo and catchy branding.

## **CVE-2015-7547**

A stack buffer overflow bug in GNU libc—the GNU Project's implementation of the C standard library. The vulnerability could allow remote code execution on victim machines, and is noteworthy due to the high-ubiquity of the vulnerable component. The bug was independently discovered by Jaime Cochran and Marek Vavruša, both of whom then worked at Cloudflare, a US-based web infrastructure company.

## **CVE-2015-3824 (“StageFright”)**

One of several vulnerabilities in the mediaserver component of android OS, all due to integer overflows and underflows. The vulnerability allowed attackers using a malformed MMS message to install arbitrary applications on victim Android phones without requiring any user interaction—only requiring the victim's phone number. Noteworthy because of its obscenely wide-reaching attack vector and its bypassing the careful habits of even the most security-conscious user, this vulnerability was discovered by Joshua Drake, then a researcher with mobile security firm Zimperium and formerly a lead exploit developer for the Metasploit framework.

## **CVE-2012-4929 (“CRIME”), CVE-2014-3566 (“POODLE”), CVE-2016-0800 (“DROWN”)**

A continuous six-year-long moment where the infosec community collectively discovered that theoretical cryptographic attacks—such as oracle attacks, downgrade attacks and precomputation attacks—were actually applicable to real-life cryptography, and namely SSL. We already have an extended write-up on the theory and practice behind these attacks and the people who uncovered them, titled [Cryptographic Attacks: A Guide for the Perplexed](#), and so we won't replicate it here.

## **CVE-2008-4250**

A buffer overflow vulnerability in MS-Windows' [server service](#) (don't look at us, we don't name these; it apparently deals with file and printer sharing). This vulnerability allowed attackers to execute arbitrary code on victim systems. The remaining details of its discovery are hazy, and seem to have involved fully working exploits being found in the wild by Microsoft, who rushed to fix the problem and create an emergency patch. This vulnerability was mostly infamous for being a main infection vector for the conficker worm, one of the widest-reaching worms ever which is estimated to have infected millions of machines.

## [CVE-2008-1447](#)

A bug in the DNS protocol—that’s a grim opening already; not “an implementation of” the DNS protocol, the DNS protocol itself. The bug allows an attacker to cache-poison a DNS server, that is, make it believe that a domain name corresponds to an attacker-crafted IP address, rather than the true IP address, and then pass on this misinformation to clients. This is done by abusing the fact that DNS was not designed for security: When a DNS server sends a request to its fellow server to retrieve the correct IP address for some domain, its only method of verifying the response is that it contains the correct Transaction ID (TXID) that the server sent originally. At the time these TXIDs had 16 bits, so an attacker attempting to forge a DNS response had a one-in-65536 chance to guess the TXID correctly. The attack was able to induce a DNS server to make these requests for the IP of a specific domain again and again, tens of thousands of times, until the attacker—by pure chance—finally nailed the correct TXID (in fact, there was a mitigation in place to prevent this—servers were supposed to only update their internal cache with a new IP address for a given domain once a day—but the attack bypassed this by querying many different subdomains: `aaaa.domain.com`, `bbbb.domain.com` and so on; responses to such DNS queries were allowed to contain IP addresses for the main domain.com). Back in 2008, Glenn Fleishman and Rich Mogull at Macworld [explained the issue in our favorite style](#):

Alex is set up with a blind date named Charlene. But Alex has an enemy named Beth. Beth finds out that Alex is supposed to meet someone at Cafe Depot for coffee at 12.10 p.m., but doesn’t know the blind date’s name. Beth sends 50,000 women to the cafe—rather crowded, now—all with different names. ‘Hey, Alex, I’m Alexis, aren’t I here to meet you?’ ‘Hey, Alex, I’m Zelda, aren’t I here to meet you?’ [...] If Beth’s hired hand named Charlene meets Alex before his real blind date, the next thing he knows, he’s been slipped a mickey, and wakes up in a hotel room with a scar where his kidney was, his wallet missing, and a whopping room service bill.

This vulnerability was discovered by the late Dan Kaminsky, a US security researcher, also known for his work on enumerating victims of the Sony rootkit and of the Conficker botnet. The attack was eventually staved off by using a kludgy hack to increase the effective size of the TXID so that the attacker’s chances of guessing the correct TXID become negligible, even if they make many attempts.

We could stay here forever reciting vulnerabilities and the names of many (many, many, many) vulnerability researchers active today who have uncovered and continue to uncover them, but we hope that this sampler gave you somewhat of an idea of what vulnerabilities are, who finds them and how.

# Q & A

with Sagi Tzadik,  
*Discoverer of SIGRed*



Every one of the names mentioned above, and many who weren't, would have made a great interview subject; but we only see one of them at the same group meeting every Wednesday. Therefore, please enjoy the following short Q&A with Sagi Tzadik, who discovered the SIGRed vulnerability (CVE-2020-1350).

---

**Q.** How did you get into vulnerability research?

**A.** Originally I was introduced to the infosec field via the Game-Hacking community. I remember 10-year-old me examining cheat codes for a game called [MapleStory](#), which were written in assembly, and wondering what was going on behind the scenes. Then I naively searched for materials online about being a “hacker” and found information about web-hacking (SQLi, XSS etc), perhaps because this was most accessible.

---

**Q.** What technologies and skills were the most important milestones for you? Any favorite study resources (courses, textbooks, tutorials) to recommend?

**A.** I think that the core of the infosec community is the question “How does it work?”. This is why I think that the ability to do a proper code review, debug and reverse-engineer to some degree are vital skills for vulnerability research. You should also be familiar with the specifics of the platform that you are researching (be it PHP or Windows)—because you should be able to identify use of bad practices—“Learn the rules like a pro so you can break them like an artist”. While I learnt reverse-engineering from old sources like [Lena151](#)—today there's much more modern stuff out there, like [LiveOverflow](#) which I would definitely recommend! If you are more into written materials I will shamelessly plug the very CPR blog you are reading. These days I consume most of my updates from Twitter, [/r/netsec](#) and following GitHub accounts.

---

**Q.** How did you end up researching Microsoft DNS server specifically?

**A.** As I said, I come from a web-hacking background so DNS is quite familiar to me. I also found it weird that SMB and RDP received so much attention while DNS can be used just as well in order to compromise an organization, so I decided to have a look there.

**Q.** How did the investigation that led to SIGRed go? What challenges did you face?

**A.** The first challenge was to reverse-engineer the binary and understand the complete flow—since the server receives an incoming query until it sends its answer. Then I had to overcome multiple obstacles such as:

- How do I make the target DNS server parse arbitrary responses that I have control of, even though it does not trust me?
- How can I send extremely large responses? (because UDP/DNS is size-limited to 512/4096 bytes)
- How can I fit even more data inside these large responses (64KB is not enough to trigger the bug)?

I found solutions to all of these problems by reading RFC documents (the most I read in my entire life) and reverse-engineering.

---

**Q.** What are your favorite and least favorite parts about vulnerability research?

**A.** My favorite part is definitely the dopamine hits. That feeling when you manage to successfully find a bug after weeks of investment is irreplaceable. There's also this feeling of "I may be the first one to ever notice this" and "I could potentially hack XYZ servers if I wanted to". The least favorite part in my opinion is the realization that not every bug is a vulnerability and sometimes you will not find a bug because either it does not exist or you simply missed it, and there's nothing you can do about it.

---

**Q.** What do you have to say to a newbie considering a career in your field?  
How would you describe your work to them?

**A.** I usually describe my work to my friends as "Glorified QA". At the end of the day I am looking for bad practices but in source code that is not my own. If you consider a career in this field, I would recommend ensuring that you are really passionate about it. This is hard work, very time consuming and you are likely to face a lot of failures while doing it. But with every failure, the next success becomes sweeter.

---

**Q.** What do you think are the most important challenges facing vulnerability researchers today? How will the landscape change?

**A.** Exploit mitigations do work. They do make our life harder. 10 years ago it would take only a single bug to exploit a browser. These days it takes much more than that, to the point that companies offer a ridiculous amount of money for a stable exploit. Also, companies are now more security-aware, which I am sure eliminates a big portion of the bugs. I mean, 10 years ago not a lot of developers were writing tests for their code, but this is definitely a more common practice now.

“SUCCESSFUL EXPLOITATION WILL BECOME MORE DEMANDING, AND I WON’T BE SURPRISED IF MANY IN THE FIELD DON’T KEEP UP”

## The Long Game and The Conclusion

Where is the field of vulnerability research going? Some time ago we asked our security research tech leader, Eyal Itkin, about this. Eyal is himself responsible for [several dozen coordinated vulnerability disclosures](#), as well as [a mitigation that was integrated into common implementations of the C standard library in 2020](#). He answered, with some apprehension: “successful exploitation will become more demanding, and I won’t be surprised if many in the field don’t keep up”.

Eyal’s answer echoes Sagi above, who noted the incessant march of ever-more-pervasive mitigations. As attackers have a more and more difficult time attacking garden-variety applications, or even accessing the code that will process their input, they must show ever-increasing proficiency and creativity when choosing and analysing targets. Sometimes this means recognizing a component

that has been given an unduly small amount of attention; other times, it means wrestling with code several layers of abstraction down from the obvious attack surface, the author of which made an active effort that it never be read. In his 2018 SSTIC talk [Closed, Heterogeneous Platforms and the \(Defensive\) Reverse Engineers’ Dilemma](#), [Halvar Flake](#) laments this latter trend as not the great win for defenders that it might seem at first. “Device and OS vendors misunderstand the iterated nature of security games [...] commercial attackers pay [the] cost to build an infrastructure [for analyzing software] once, defenders have to pay it again and again. [...] All other platforms [than Linux] have gotten harder to debug, harder to introspect [...] these ‘security’ measures have become like DRM: Primarily an inconvenience to the good guys. [This is a] net loss for overall security under any reasonable set of assumptions.”

While advances in the art of frustrating attackers have certainly been made, the tools available to attackers have been evolving, too. We mentioned before the relatively new ascendancy of the “fuzzing” paradigm, and its surprising strength. Another venue for more sophisticated vulnerability hunting is the heavily theoretical field of Symbolic Execution—that is, automated analysis that reasons deductively about code to find input that will induce it to behave unexpectedly. This technology is not as ripe for use as fuzzing. In its most naive form, it struggles with truly imposing roadblocks (e.g. analyzing 20 conditional statements requires keeping track of possible states), and its more applicable form is basically a “smarter fuzzing” where “interesting” perturbations to existing input are deduced, instead of searched for at random. As far as we know there is no Symbolic Execution tool that will allow the average researcher a comparable experience to a fuzzer with regards to ease of setup and yield of vulnerabilities. Still, the technology is theoretically there: Wikipedia lists [almost a dozen different symbolic execution tools](#), even if “many tools [...] have not been made available to the public at large”, and one such tool is Microsoft’s SAGE, which [according to the company](#) “has found many previously-unknown security vulnerabilities in hundreds of Microsoft applications, including image processors, media players, file decoders, and document parsers [...] [as well as] roughly one third of all the bugs discovered by file fuzzing during the development of Microsoft’s Windows 7”. This was already happening in the late 2000s (and reported in the paper above, dated 2013). One can only assume that as more time passes, this technology will improve and proliferate.

Vulnerability research is already a practice-oriented, rather than theory-oriented, field. We speak from bitter experience when we say that one week of access to a setup that already works, or one conversation with someone who’ll hand you the right script and point you in the right direction, can be easily worth a mountain of familiarity with the fundamentals and half a year of reasoning from first principles. The above-described race of ever more complicated and specialized tools, targeting ever more obtuse and inaccessible code, seems poised to amplify this feature of the field further. We can only hope this trend will be tempered by the ancient tradition where one practitioner magically stops what they were doing for a minute, even though they have no apparent incentive to do so, says “wait... why is this so difficult? I bet it could be made much less difficult”, and a year later every wide-eyed beginner gets to do some research the easy way. 30 coordinated disclosures later, if enough users are dragged kicking and screaming to apply their software updates, maybe the next Wannacry incident doesn’t happen. We are sadly a long way away from a world where everyone is Excellent to Each Other, as per the CCC banner, but this would be a nice place to start.



**Worldwide Headquarters**

5 Ha'Solelim Street, Tel Aviv 67897, Israel | Tel: 972-3-753-4555 | Fax: 972-3-624-1100 | Email: [info@checkpoint.com](mailto:info@checkpoint.com)

**U.S. Headquarters**

959 Skyway Road, Suite 300, San Carlos, CA 94070 | Tel: 800-429-4391; 650-628-2000 | Fax: 650-654-4233

[www.checkpoint.com](http://www.checkpoint.com)

© 2022 Check Point Software Technologies Ltd. All rights reserved.