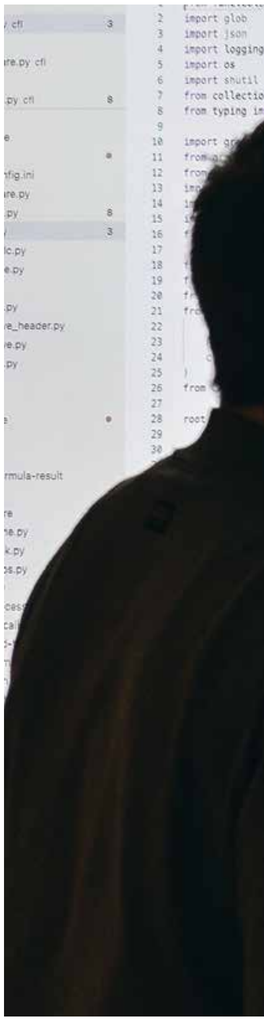
Entry point for cybercriminals: software generally has security vulnerabilities. The automated testing procedure developed by a Max Planck team effectively and efficiently checks programs for them.

ILLUSTRATION: ALESSANDRO GOTTARDO

PROGRAM VULNERABILITIES

TEXT:
THOMAS BRANDSTETTER

Attacks on software not only create billions of dollars in damage, but also threaten the privacy of users. Cybercriminals infiltrate programs through security holes. Marcel Böhme and his team at the Max Planck Institute for Security and Privacy have undertaken the task of closing entry points to attackers – and their approach has even caught the attention of companies such as Google.



Programming is a creative process. It starts with a programmer having an idea to implement a desired feature and ends with working code. But it is by no means certain what will work. The devil is often in the detail, and it can prove to be a serious threat. The Heartbleed bug, for example, which resulted in access data to numerous online services being made public in 2014, was based on a security vulnerability in software with a very straightforward task: the small program was called Heartbeat and was designed to solve the problem of a browser sometimes continuing to send encrypted data when surfing the internet or banking online via a secure connection, even though the protected connection had long since been terminated. Heartbeat makes it possible for the browser to ask the server whether the connection is still secure. To do this, the software regularly sends a string of characters, including the number of those characters, to the server and expects the same string of characters as a reply. The developer of course assumed that the browser would always give the correct number of characters. But counter to standard practice with other software, he did not build in a mechanism to check this. The attacker exploited this loophole and manipulated Heartbeat to make it send a short string of characters indicating the maximum length. Then, when the server reads the number of characters out of its memory, it copied considerably more data than the original combination of characters – including sensitive information. Experts refer to this case as memory corruption.

All too often, the programmer's intention to solve one particular problem is subverted by malicious hackers; the hacker asks themselves how they can use the software's code for their own purposes. Attacks by cybercriminals who encrypt all files in order to demand a ransom for their release are also notorious. These types of ransomware attacks are responsible for most of the economic damage caused by cybercrime. According to the industry association Bitkom, this totaled more than 220 billion euros in Germany alone in 2021.

The problem of IT security is further compounded by the fact that modern software systems are seldom developed by a single person or even by a single company. Rather, they are often assembled from a variety of components that come from different sources. And each of these components in turn consists of small individual contributions from independent programmers who have varying approaches to security. "There are an awful lot of small open-source software systems that were perhaps developed by someone in their spare time one afternoon, but over the last 20 years have become incredibly critical and fundamental to our digital economy," says Marcel Böhme, head of the Software Security Group at the Max Planck Institute for Security and Privacy. But, aside from the original developers, no one feels responsible for the security of all these individual components. And after such a long time, some of the developers are simply no longer interested in further developing their program. Despite this, there are still some advantages to publicly available software elements, especially in terms of security, as many experts can check them. This is another reason why companies such as Google use open-source code. Google is now working with Böhme and his team to detect security vulnerabilities in its own software.

SUMMARY

Software developers need to make sure that their programs are not vulnerable to attack. Software that is not tested for security vulnerabilities often has critical errors.
Researchers at the Max Planck Institute for Security and Privacy have dramatically accelerated automated vulnerability scanning using greybox fuzzing.
Companies such as Google, Bosch, and Oracle Labs are already using the method and discovering new bugs all the time.

One of the major problems of software security is that the chain of program components is only as well protected as its weakest link. And this leaves even large commercial systems vulnerable to attack. In some cases, all it takes for a resourceful hacker to take over the entire system is to identify a single poorly protected component. "This is often because this aspect was not very important at the time the program was created," explains Böhme. "Many of the major security vulnerabilities we find in systems that are in use around the world today can be traced back to these kinds of 'small' vulnerabilities." Memory corruption is particularly critical here, he said. Not only can it be exploited to spy on and steal data – as in the case of Heartbleed – but it can also be used to smuggle commands into a program that, in the worst case, allows an attacker to take control of the computer. In the same way that more can be read from memory than is specified, more can also be written to it than is intended by a program – provided that the software is not programed to check the specifications for the amount of data requested or transferred. In order to detect problems like this in the program code, current systems, even when already in use, are extensively tested for critical holes over weeks, months, and sometimes even years – and repaired if necessary through security updates that are distributed with priority. This is also true of software that many people use at home. If, for example, a security vulnerability was to emerge in Google Chrome that could be used to spread malware, countless users around the world would be affected and could become victims of ransomware attacks. "But personal data, passwords, or browsing be-

Many hands make light work: the more people check programs, the more likely it is that security-relevant errors will be discovered – especially when experts like those in Marcel Böhme's team are the ones doing the checking. That said, automated tests speed up the search.

"Greybox fuzzing combines the best of both worlds."

MARCEL BÖHME

havior, for example, could also be stolen," Böhme warns. "Fortunately, the likelihood of that happening with Google Chrome is very small," he says. "After all, Google has many ways of protecting its systems."

There are a number of approaches to finding dangerous flaws in a program. The simplest approach is for people to take a close look at the software and look for bugs. "But machines that do this automatically increase the chances of success," explains Böhme. With automated methods, a distinction is made between static analysis and so-called "fuzzing." Static analysis methods start by examining a program's code and using it to create a model that describes its behavior. Marcel Böhme illustrates this with a comparison to biology. In one branch of this discipline, bioinformatics scientists might, for example, replicate a cell on a computer, simulating the interaction of its various components according to biological rules. "In a similar way, you can also create a model of a program," Böhme explains. "This is done by simulating its behavior based on the syntactic and semantic rules used to write the software." This model is then used to try to predict all conceivable inputs and ultimately prove that none of them can lead to a critical error.

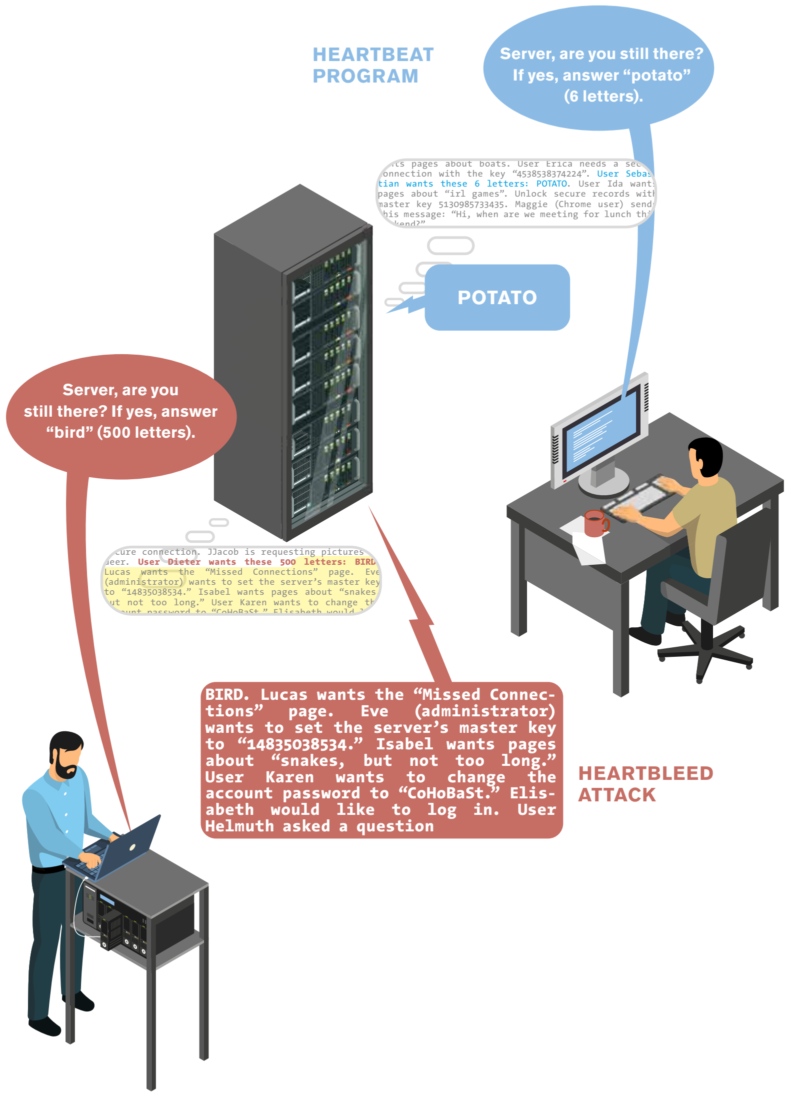
⟶

Just like the cell replicated on the computer, however, the model of a program is missing its real environment – which in the case of a program is all the other programs with which the tested program communicates.

Systematic random input

By contrast, fuzzing – which is what Böhme and his team have devoted themselves to – involves running the program under real conditions and subjecting it to as many randomly generated inputs as possible. "Fuzzers essentially simulate a user who is not doing what the programmer envisioned," Böhme explains. In its purest form, black-box fuzzing, no meaning is attributed to the interrelationships in the code of the program under study, and errors are detected solely by random input. Blackbox fuzzers, however, also frequently test program parts that are executed during use much more often than necessary, which is not very efficient. But what's worse: the black box also rarely tests program parts that are seldom in demand, or in the worst case it does not test them at all, meaning that security vulnerabilities in them remain undetected. This problem can be solved with whitebox fuzzing: here, the code is also analyzed and, similar to the static methods, converted into a formal model, which is then systematically examined down to the last detail. "This is very effective, but takes far too long for modern programs," says Böhme.

Consequently, at least at the beginning of debugging, randomly generating input values is more efficient than systematically exploring the program's behavior. However, as testing progresses, the whitebox approach gains ground because of its ability to learn and not have to revisit behavior that has already been tested. The black box approach, on the other hand, does not care whether or not it has already tested a particular behavior. "As a scientist, it is somewhat counterintuitive for a bombardment of strictly randomly generated inputs to yield better results than a deep analysis," Böhme says. "So at the end of my PhD, I became interested in the question of whether this behavior could be explained." Besides security, another factor to consider is the amount of time required. While a whitebox approach creates just one or two inputs per second, a blackbox fuzzer can easily create hundreds of thousands of randomly generated inputs per second. "We have figured out which approach works best under which conditions," Böhme tells us. "That then led us to greybox fuzzing, which combines the best of both worlds, in a sense." A greybox fuzzer generates input just as fast as a blackbox fuzzer, but also uses additional feedback about the parts of the program that have already been executed, just like a whitebox fuzzer. In doing so, greybox fuzzers avoid the repetitive testing of the same software elements, which slows down the whole process. At the same time, they also ensure that parts of the software with niche functions aren't overlooked.


HEARTBEAT PROGRAM
Server, are you still there? If yes, answer "potato" (6 letters).
ts pages about boats. User Erica needs a sec onnection with the key "4538538374224". User Sebastian wants these 6 letters: POTATO. User Ida wants pages about "irl games". Unlock secure records with master key 5130985733435. Maggie (Chrome user) sends his message: "Hi, when are we meeting for lunch thi kend?"
POTATO
Server, are you still there? If yes, answer "bird" (500 letters).
cure connection. JJacob is requesting pictures ueer. User Dieter wants these 500 letters: BIRD Lucas wants the "Missed Connections" page. Eve (administrator) wants to set the server's master key to "14835038534." Isabel wants pages about "snakes, ut not too long." User Karen wants to change th ount password to "CoHoBaSt." Elisabeth would l
BIRD. Lucas wants the "Missed Connections" page. Eve (administrator) wants to set the server's master key to "14835038534." Isabel wants pages about "snakes, but not too long." User Karen wants to change the account password to "CoHoBaSt." Elisabeth would like to log in. User Helmuth asked a question
HEARTBLEED ATTACK

Manipulated echo: the Heartbeat program checks for a secure connection by regularly requesting changing character combinations, each time indicating the number of characters. In 2014, cybercriminals tricked the software into retrieving significantly more characters from a server's memory than the requested response. This was how they tapped into sensitive data.

DIAGRAM: BASED ON HTTPS://XKCD.COM/1354/, ISTOCK

The greybox fuzzer is a great error hunter: "When a piece of software is subjected to fuzzing for the first time, we find an average of two to three bugs, including security vulnerabilities, per day," says Böhme. "After a few weeks, this reduces to three or four new bugs a week and then stays constant, since new bugs are introduced all the time." The combination of efficiency and security is also winning over tech companies: at Google alone, 100,000 computers are now devoted to running a greybox fuzzer and using it to test over 500 software projects around the clock.

The fuzzers developed at the Max Planck Institute are exclusively open-source applications, which means that they are freely available on the Internet. "By taking this approach, we're also making it available to small-scale programmers to help them troubleshoot their own programs," Böhme says. Furthermore, larger open-source projects are also being scanned. Just recently, for example, Böhme's fuzzers uncovered a serious security hole in OpenSSL, a free software program for encrypted communication in browsers and e-mail applications. "The security hole our team found would have allowed an attacker to take over computers sending encrypted e-mails," the computer scientist explained.

Security for the Internet of Things

Despite many collaborations with large companies such as Bosch and Oracle Labs, Google is still the most important collaborative partner for Marcel Böhme's research group. The American tech giant has a strong interest in the security of open-source projects, as they also represent essential components of its own products. "The cooperation with Google is interesting for us because they are the market leader in this area and have extensive resources that we would otherwise not have access to," explains Böhme. "It's a kind of symbiosis." After all, even though Google could use the freely available fuzzers anyway, the group is always close to the current state of development due to our close cooperation. Despite all the current successes, Böhme also plans to take on new challenges in the future. After all, digitalization and artificial intelligence are turning the world of data processing upside down, necessitating completely new security concepts. Industry 4.0, for example, is a trend that moves away from large, centralized computing units and toward many small devices that have various sensors and can, therefore, perform smaller tasks. "They usually have relatively little computing power, which is why security is often considered to be of very little relevance during the development of such systems," Böhme explains. These small units often exchange data with other devices, for example, to allow them to perform calculations on their own. "To ensure that the small devices in this Internet of Things work properly on a large scale, you could try, for example, to designate one of them to test all the other devices," Böhme says. The security of machine-learning algorithms, or artificial intelligence, is also becoming an increasingly important issue in society. "These systems operate on completely different principles than classical computer programs," says the researcher. "But unfortunately, we don't yet have any techniques for ensuring that the likes of an AI assistant actually does what it's supposed to do." Böhme believes that changing this is an important task, one that he and his team would like to devote more time to in the future.

www.mpg.de/podcasts/sicherheit (in German)

PHOTO: FRANK VINKEN FOR MPG

Race against the attackers: Marcel Böhme and his team are working to make the Internet of Things and artificial intelligence secure. He discusses a new idea with Kirandeep Kaur to close loopholes for cybercriminals.

GLOSSARY

FUZZING
(derived from fuzzy) refers to automated testing procedures that check software for security vulnerabilities using random input. A distinction is made between purely random blackbox fuzzing, whitebox fuzzing with additional model analysis, and greybox fuzzing, in which a program is systematically tested with mass random inputs.

STATIC SOFTWARE ANALYSIS
attempts to prove mathematically that no possible input can lead to a critical error in the program.