Introduction to Software Security
Chapter 1.3: Thinking Like an Attacker
Owning the Bits

| Loren Kohnfelder	Elisa Heymann	Barton P. Miller
loren.kohnfelder@gmail.com	elisa@cs.wisc.edu	bart@cs.wisc.edu

DRAFT — Revision 1.2, Feruary 2019.

Objectives

- Learn and understand the basic vocabulary that describes an attack.
- Make these ideas more concrete by learning about a classic type of attack.
- Introduce the thought processes of an attacker, including the idea of "owning the bits" in a program.

WARNING: If you want to experiment attacking systems, set up your own instances with pretend data and always get explicit permission from the system owners. Unauthorized attacks, even harmlessly done with the purest intentions, may incur serious criminal penalties not worth the risk.

An Exploit through the Eyes of an Attacker

Let's think about an attack the way an experienced attacker would in order to gain a solid understanding of how adversaries think and act. Attackers view software almost in a mirror-opposite way than we do. The best security software engineers are able to flip back and forth between both ways of thinking, playing a kind of mental chess game between opposing forces.

Programmers writing code have in mind what the program should do and then express that as code using the language facilities and component libraries available. In practice, the mapping between intended function and the actual functioning of the resultant code are almost never exact, and best practice demands thorough testing and careful debugging in order to ensure that these are closely aligned.
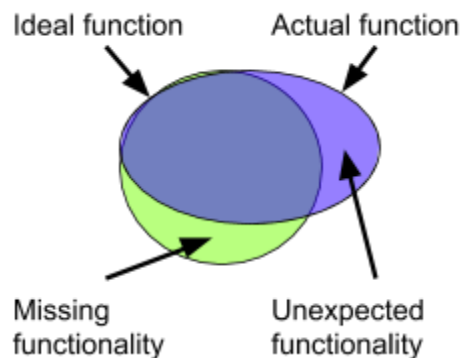
In the diagram above, the semi-transparent green circle and blue oval symbolize the difference between an ideal software goal and the actual functioning of a real implementation that approximates it. When we say "all software has bugs" we mean that these will never be identical. The green circle is all ideal functions that the programmer has in mind and is endeavouring to build. The blue-green overlap represents the substantial portion of the intended function that is already working.

Even with excellent test coverage, debugging effort will focus on the intended functionality that is not yet working, in effect stretching out the blue oval to include more of the green circle. Unfortunately, code changes intended to implement the desired functions can "overshoot" and do other unintended things as well — something like the oval extending past the green circle. Alternatively, a change can easily distort the far side of the oval as a consequence causing part of the green circle already covered to now be exposed (this is the effort of code changes causing new bugs as a side effect).

Note that programmers don't think about or even perceive the part of the blue oval sticking out on the right — that's additional functionality in the code itself that was never intended in the first place. With the programmer's attention firmly focused on the lower left side of this diagram as explained above (covering more of the green) it's psychologically nearly impossible to be fully aware of this whole other side of the program at the same time. Also note that functional testing is unlikely to reveal this extra unintended functionality. This is why it's valuable to come back and review work with your "security hat on" when you are thinking like an attacker. Of course this diagram is simplified to demonstrate the core point. In practice, the "shape" of the functionality of complex programs is multidimensional and intricately detailed so the places where intended and actual functionality differ are many and varied.

These discrepancies are exactly where attackers focus, and as explained above, programmers tend to be blind. If code is written to work only up to some limit then attackers want to exceed that limit and determine what happens (a buffer overflow is one common example of this principle). Should some rare combination of inputs causes quirky behavior, attackers want to poke at this to see if they can weaponize it. When code is fragile, attackers attempt to break it and then see if the "sharp edges" can be used to cut through defenses or somehow cause further damage.

At first attackers may act like a bull in a china shop simply trying to break everything in sight. This stage can be a learning process, either finding fragile points on the attack surface, or confirming that the code is solid and perhaps deciding to poke around elsewhere. In time the attackers learn more about what they are up against and begin to focus efforts toward making a specific exploit to achieve their goals.

When we approach an attack from the attacker's perspective, it leads to an alternative way to define an exploit from what we introduced previously. This definition comes from the intelligence community. In this context, we think of an exploit as a manipulation of a program's internal state in a way not anticipated (or desired) by the programmer with the goal of creating unexpected behaviors in the code.

Owning the bits

Let's look at a concrete example of an attack to bring all of this strategic perspective into focus. Consider a public web server that accepts data entry from a web form and copies the contents into memory for processing. Anyone on the internet can submit form data so the attacker has a direct path into the web

server process, and whatever characters they send via the form are reliably copied into the corresponding memory buffer. Since the attacker can reliably cause whatever string they choose to appear in the server memory we say that the attack **owns the bits** and so long as the string is harmlessly processed there is nothing wrong with this at all.

However, if the code is poorly written — let's say the lazy programmer made the memory buffer 100 bytes long figuring nobody would ever enter anything that long — then by sending ridiculously long web form field character strings the attacker now owns the bits of that memory buffer and arbitrarily beyond (including potentially all kinds of juicy memory locations). This is a classic vulnerability called a buffer overflow, to be explained in detail below, and it's an example of the kind of unintended functionality that attackers love to discover.

Such an attack path consists of a chain of events that always starts at the **attack surface** which is the externally facing interfaces an attacker can access, e.g. interfaces on the public internet, peripheral devices the attacker can use, an exposed private network tap, and so forth. Software operates on this attacker-controlled input (usually a complex chain of events) and in the case of a vulnerability leads to many actions and state changes. Eventually, from the perspective of the attacker's exploit goal, something bad happens somewhere. The set of all ultimate actions the attacker can cause to happen via these various attack paths is called the **impact surface**.

A Real World Exploit

To make these concepts clear, let's look at a non-software real world attack from this perspective: how might an attacker rob a bank?

As described above, attack paths consist of chains of events that leverage weaknesses (specifically, vulnerabilities) to exert some control to the attacker's ends, and reach down to the impact surface where actual harm is done. To fully understand the attacker mindset, let's think about attacks in terms of the real world scenario of robbing a bank.

Using the model described above, the attack consists of a specific sequence of events:

- Attack Surface: where the attack can begin that the robber has access to.
- Attack Path: a sequence of actions the robber takes.
- Vulnerability(ies): one or more weaknesses that can be exploited.
- Impact Surface: malicious results that follow from the above chain of events.

Attack Surface	Path of Attack	Weakness(es)	Impact Surface
Front door	I request cash … → Teller opens cash drawer … → Teller distracted by my accomplice … → Reach over counter and remove cash.	1. Teller turning away while drawer open. 2. Counter open and too narrow.	Removing cash from teller's drawer.
Back door	I try to open the door … → Look to see if vault is open … → See if cash available … → Grab cash and go.	1. Back door unlocked. 2. Vault left open. 3. Cash not locked in drawer.	Removing cash from the vault.
Window	. . .	. . .	. . .
Corrupt teller	Takes cash from drawer … → Hide cash in backpack.	1. Weak background check. 2. Insufficient inspections.	Removing cash from drawer.

First, to begin with the most obvious approach, the robber could simply enter the bank through the front door as a customer (the most obvious part of the attack surface). Once inside, the "customer" goes to the teller counter to request a transaction, and when an accomplice distracts the teller, reach into the cash drawer grabbing some money and making a getaway. The relevant weaknesses include an unprotected cash drawer within reach of customers, the teller's inattention, lack of a surveillance, inability to lock down quickly to prevent getaway. The impact surface is the robber grabbing the cash drawer filled with money.

Next, consider the back door threat where the robber discovers that a rear entrance to the bank is unlocked and goes inside. From here the robber's plan is simple: find the vault and see if it's open; if it is, enter and take cash or anything of value. The weaknesses enabling such a heist are evident: unlocked rear entrance; leaving the vault open; cash lying around.

In a real bank there are many other possible attacks but let's look at one more where there is no bank robber involved, but instead consider what a corrupt employee might do. As a bank employee a dishonest teller could at any time discretely stash some cash in a backpack and walk out at the end of the day. This rarely happens in real banks and in this hypothetical case quite a few weaknesses would need to be present for it to be a viable attack: insufficient background check screening employees, lack of audits and inspections to ensure all transactions are accounted for, and lax business practices making such an audacious crime even possible. This is an example of an **insider threat** where a person violates trust and abuses their authorized access. Security cannot eliminate the possibility of insider attacks since bank

employees need access to money to do their jobs. Ensuring accountability through auditing and oversight mitigates the potential risk.

The Attack Surface

The attack surface is all the ways that a user (or attacker) can affect the behavior of a system. For a bank robber this would be the entire exterior of the bank: doors, windows, walls, roof, tunneling underground, or utility connections.

In a typical program there are many different parts to the total attack surface.

- Network Data: if the attacker can send data over the network that invokes the code.
- Input File: if the attacker can influence data in a file the program will read.
- Web Form Field: if the attacker can access a web page providing input to the code.
- Database Entry: if the attacker can control data in a database used by the code.
- Configuration: if the attacker can alter settings that influence the code.

The attack surface also includes everything the attacker can externally observe about the system. For a web server this includes all public pages, including scripts and error responses. Open source software allows attackers to study the details of the code that can reveal unexpected functionality and hints about how to access and influence the system in new ways.

Direct and Indirect Attacks

Direct attacks are when the attacker provides input at the attack surface that ripples through the system triggering a vulnerability to cause harm. There are **indirect attacks** as well, where a series of actions cause state changes, and those in turn cause other things, eventually causing harm as a result of several steps (something like a [Rube Goldberg apparatus](#)).

An example of an indirect attack will make the distinction clear. Consider a public web service that accepts requests and validates all inputs rejecting invalid inputs, responding with an error page and logging the details for later analysis. A clever attacker crafts a malicious request and receives the rejection error response, however, as a result of this input the log entry for this event is malformed and that confuses the logging system. As a result, even though the request was rejected, the attacker has managed to own bits in a destructive way in the log files.

Eventually the daily log analysis job starts up and reads through the day's log files preparing a report. When it gets to the malformed log entry this triggers a vulnerability and now the attacker has gotten into the log analysis job in a powerful way, positioned to do harm from here.

This example illustrates the importance of understanding the attack surface, and what attack paths extend from it. On casual consideration one might consider the log analysis job immune from external attack because it does not accept any requests from the public internet and is only executed by a timed script by system operators — but this would be wrong as the example clearly shows.

Direct attacks, by contrast, are the most common and obvious because there is a direct chain of causality from attack surface to vulnerability (even though in large systems these can be extremely complex and difficult to trace through completely).

Following the Flow of Your Code

Attack paths propagate from attack surface to impact surface through a combination of data and control flow that forms a chain of events the attacker may have more or less influence over.

Control flow is the branch and call paths as affected by inputs originating at the attack surface. Data flow is the propagation of data and derived values also originating at the attack surface. Let's examine closely how this works in practice with real code.

Suppose that variable **i** came from the attack surface, and this influences the value of **a** by selecting which element of the **buffer** array is chosen. In a following statement, the value of **a** is used to compute **b**, extending the reach from the attack surface further. Later on, the value of **b** indexes the **stringtable** array and the resultant character string is printed. In this indirect way and subject to numerous limitations, the attacker has a program flow attack path from **i** influencing the printed string.


```
...
a = buffer[i]
...
b = a mod n;
...
print(stringtable[b]);
```
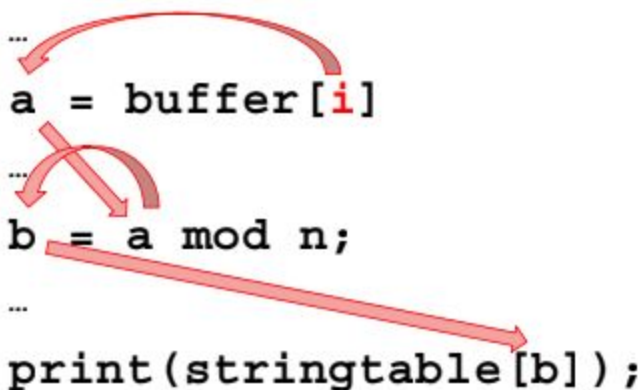
Sometimes a combination of control and dataflow make the attacker's influence harder to see.

```
if (buffer[i] > 10)
    val = 3;
else
    val = 25;
```

In the code above, the value of **i** from the attack surface (as an index to **buffer**) influences the value of the variable **val** even though it's not directly assigned — the two variables don't even appear in any statement together (and could be distantly separated in more complex code). The takeaway here is that program flow can easily extend the reach of a potential attacker from even a minimal attack surface into a

much larger impact surface than you might expect. Minimizing the impact surface can be an important defensive strategy to make code more secure.

The Impact Surface

The *places in the code* where the exploit actually takes effect is called the **impact point**. The collection of all such impact points is called the **impact surface**: it's what the attackers are trying to get at and often the most obvious place that we have to defend in the code.

Examples of impact points include:

- Unconstrained execution (e.g., executing a shell). The attacker manages to execute a command that results in executing a shell where the attacker can run any command they want. The attacker could achieve that by owning the bits of the variable that contains the name of the executable file that is the argument to the exec (or system) call.
- Privilege escalation. The attacker (working in a context with user-level privileges) carries out actions that allows them to operate with higher privileges than intended. An attacker could achieve that by owning the bits of a variable used for authorization.
- Acting as an imposter. An attacker can impersonate a valid user of the system, such as by guessing a weak password. In doing so the attacker can own many bits they are not otherwise authorized to access.
- Forwarding an attack. An attacker could perform a phishing attack to get some personal information from a user. That information could be then used by the attacker to act like an imposter, as described in the previous item.
- Revealing confidential information. An attacker can gain access to the bits of variables that contain confidential information.

Tying it all together: we think of an attack starting at a point on the attack surface, based on an input from the user. This input causes a chain of events to help, following the flow of control in the program, until it reaches the place where some "bad thing" happens, which is a point on the impact surface. An attacker tries to visualize what is happening in the code, trying to understand how a given input controls the state of the program, i.e., which bits do they own, leading the flow of control to an effect point on the impact surface.

The Classic: A Stack Smash

Let's see an example of owning the bits in the classic stack smash attack. Consider the C code below, where we have a buffer of 100 chars and two integer variables, both allocated on the stack

```
int foo()
{
  char buffer[100];
  int i, j;
  ...
```

```
  gets(buffer);
  …
  return(strlen(buffer));
}
```

When this function is invoked, the stack contains the return address, the buffer of 100 chars, and the integers i and j, as shown in the graphic below on the right.

This program calls **gets**, which has no way to limit the size of the user supplied input. An attacker could provide as input a string of 100 chars followed by an evil return address. That evil string will overwrite the return address as we see on the left in the figure below. While it is expected that the user will own the bits in the buffer, it is neither expected nor safe, that the attacker as user will own the bits of the return address which enables arbitrary control of program execution.

This was a simple example of owning the bits, as the input directly modified the target internal state without any dependencies on complex control or data flows. Also the attacker owned all the target bits, so had complete control over the destination address.

In general, misusing pointers and strings is a serious problem in C and C++. We will talk about this topic in detail in the Pointers and Strings module.

Nowadays stack smash attacks are more difficult, as operating systems and compilers use techniques that try to prevent that kind of attacks. Example of those defensive techniques are randomization of memory locations of code, stack, and heap. Those are called address space layout randomization or ASLR.

There are also Internal consistency checks,such as heap guards or stack canaries. And there is OS/processor memory protection such as WX (meaning that a page is either writable or executable "W xor X").

Even so, attackers still find clever ways to circumvent these mechanisms. We will talk more about these defenses in the Processor and System Defenses module.

Attacker tactics and motivation

Attackers of all types are out there, and we are now all connected by the internet so this is not a theoretical concern by any means. Skill levels range from inexperienced people just playing around to highly trained professionals working in concert with nation-state intelligence operations with advanced technical capabilities backed by abundant resources. Attackers have a rich set of powerful tools and increasingly use automation to do their work.

It is not necessary to spend time trying to imagine who might attack your systems or even guess what might motivate anyone to do such a thing. Some attackers work for pay or profit, others for political or personal motives, while others are simply fascinated by the technical challenge involved. The point here is that attackers are varied and ultimately unknowable, but are not to be underestimated.

> *"I don't worry about what my opponent is doing." — Maria Sharapova*

Attackers work in anonymity and can be quite patient. So long as an attack cannot be traced back to them, they are free to work at leisure until they reach their target. Our goal is to make that target as small as possible and difficult to hit. Through design, coding, assessment, and use of tools, this course introduces you to techniques that will help make your software more secure.

Summary
- Learned how an attacker thinks about owning the bits of your program.
- Introduced important terms and concepts, including attack surface and impact surface.
- Learned an example of owning the bits with a stack smashing attack.

Exercises
1. Take a simple ("Hello world" or similar) web server and creatively "own some bits" in the server.
2. Write a small program with more than a trivial attack surface. Put on your "security hat" trace the program flow to discover the most impact surface that you can. Probe from the attack surface trying to reach the impact surface. (Better yet, do this with a partner where you each write a program, swap them, and attack the other's code.)

3. Explore the outlines of one of the most sophisticated attacks of all time.
   https://www.zdnet.com/article/infographic-how-stuxnet-supervirus-works/
4. Read the classic (three page) paper "Reflections on Trusting Trust" to learn how a malicious compiler modification can be hidden with no trace visible in the source code.
5. Challenge: modify an open source compiler using the above technique but *do not release it!*