Lecture Notes on Safety & Semantics

15-316: Software Foundations of Security & Privacy
Matt Fredrikson

Lecture 2
January 15, 2026

1 Introduction

We begin by talking about safety, and in particular one important class of safety properties: memory safety. Mistakes that programmers make having to do with memory safety have been responsible for a lot of security vulnerabilities over the years, and we will discuss some of the different types of memory safety errors that lead to vulnerabilities in common programs.

Our goal is to write code that does not have these vulnerabilities. There are many ways that have been proposed to avoid memory safety errors: type systems that ensure that memory is always used correctly, tools that check for memory safety errors at compile time, runtime systems that detect and prevent errors, and many others at various parts of the software and hardware stack. To understand how these work, and what their limitations and tradeoffs are, we need to understand how memory safety materializes in programs at the level of precise semantics.

In the second part of this lecture, we will go into detail to understand the semantics of programs and how they relate to proving things about how programs will behave when run. We won't yet define exactly what memory safety means at this level, but we will have laid the essential groundwork to do this, as well as to start reasoning about the safety of programs in more systematic ways.

2 Safety & Liveness

Our goal in this course is to reason about security properties of programs so we can prevent vulnerabilities and fend off attacks. There are different kinds of such properties, and they require different techniques to enforce them. One way to classify

them is to think about them as properties of *traces* of a program, that is, the (possibly infinite) sequence of states or events that take place when a program executes.

Safety Properties Intuitively, a safety property means that "*nothing bad happens*" during a computation. So every finite prefix of a trace should satisfy some specification that excludes "bad" states or events. Common examples of "bad" are programs that, in C, have undefined behavior. This includes division by zero, integer overflow, double free, or accessing memory whose value is undefined. The latter is exploited in so-called *buffer overflow attacks*. An example from concurrency are race conditions between threads. Another example is a policy that requires that a principal is authorized before giving them access to a resource, in which case the "bad" thing is unauthorized access.

Liveness Properties Intuitively, a liveness property captures that "*something good happens*" during an execution. For example, a server should eventually respond to a request, or a deleted file should actually disappear and be no longer recoverable.

Liveness properties are more intrinsically more difficult to reason about and enforce than safety properties. There are also security properties that can not be captured as properties of a single trace. We will consider such a property, namely *information flow*, in the second part of the course. For now, we'll work towards an understanding of a type of safety property, memory safety, that plays an important role in secure coding.

2.1 Memory Safety

Informally, a program is *memory safe* if every memory access it performs is well-defined: it only reads from and writes to memory that it is allowed to access, and it interprets that memory consistently with the way it was allocated and initialized. In low-level languages like C and C++, violations of memory safety often fall into "undefined behavior", meaning that the language does not prescribe what happens next. This is where security comes into the picture, because it is often possible to shape what happens next into something useful for an attacker.

We'll refer to memory safety *vulnerabilities* as the security-relevant instances of memory safety errors: situations where an attacker can influence program behavior or observe secrets by triggering an invalid memory access. Let's take a quick look at some of the ways that this can arise in programs.

Out-of-bounds Out-of-bounds vulnerabilities are a very broad class of errors that occur when a program reads or writes outside the bounds of an allocated object. They commonly arise from unsafe array indexing, pointer arithmetic

errors, or integer overflows that miscompute sizes or offsets. There are numerous ways that they can be exploited, from control hijacking (e.g. via overwritten control data such as return addresses or function pointers) to reading sensitive data from neighboring memory.

Use-after-free Use-after-free vulnerabilities arise, as the name suggests, when a program continues to access memory after it has been deallocated. They typically result from confusion about ownership patterns or aliasing where one part of the program frees an object while another still holds a pointer to it. A typical exploitation pattern for UAF errors involve forcing the freed memory to be reallocated with attacker-controlled data, so that subsequent accesses operate on maliciously crafted contents.

Uninitialized memory use When a program reads memory that has been allocated but not fully initialized with defined values, then stack variables, partially initialized structs, or reused heap allocations may leak sensitive information or lead to control hijacking via attacker-controlled residual data. Although compilers, libraries, and widely-used runtime environments have done a lot to mitigate the frequency and severity of these errors, they remain a true security problem.

Race conditions These vulnerabilities occur when concurrent threads or processes access and modify shared memory without proper synchronization, leading to inconsistent views of object state or lifetime. They often manifest as use-after-free or out-of-bounds accesses when one thread frees or resizes an object while another is using it. Attackers exploit timing windows to trigger invalid memory accesses in a controlled way. The consequences include memory corruption, data leakage, and arbitrary code execution.

Each of these types of memory safety errors can manifest in countless ways. Our goal is to guarantee that none of them exist in programs that we write, but first we will need to find a way to define what memory safety is, so that we can be sure that we have addressed every possible case that falls into the scope described in this section.

3 Straight-Line Programs

We now present our small imperative programming language in stages. The development is inherently open-ended in the sense that we will introduce more constructs as our study goes on.

For the sake of simplicity we assume that all variables range of the integers $\mathbb{Z}$. We have a simple language of *arithmetic expressions* e, with the usual conventions

that we do not detail here. We use a, b, c for integer constants and x to stand for variables.

$$\text{Arithmetic Expressions} \quad e \quad ::= \quad c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \ldots$$

Since expressions contains variables, their meaning is determined with respect to a *state* that assigns integers to variables. We use ω, μ, ν to range over states and assume that they are defined on all variables. We write $\omega(x) = c$ if ω maps x to c. Then the value of expression e in state ω is written as

$$\omega[\![e]\!] = c$$

and is easily defined based on the structure of e. For example:

$$\begin{array}{lcl} \omega[\![c]\!] & = & c \\ \omega[\![x]\!] & = & \omega(x) \\ \omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!] \\ \omega[\![e_1 - e_2]\!] & = & \omega[\![e_1]\!] - \omega[\![e_2]\!] \\ \ldots & & \end{array}$$

The last two equations may look somewhat odd—we have to keep in mind that '$+$' and '$-$' on the left-hand side are pieces of syntax that form expressions while '$+$' and '$-$' on the right-hand side are the mathematical operations on integers. Other operations are defined analogously.

Programs are denoted by α and β and we start here with two simple constructs: *assignment* $x := e$ and *sequential composition* $\alpha \,;\, \beta$.

$$\text{Programs} \quad \alpha, \beta \quad ::= \quad x := e \mid \alpha \,;\, \beta \mid \ldots$$

The meaning of a program is a *relation* between the *prestate* and *poststate* of its execution. It is a relation instead of a function because we would like to accommodate nonterminating programs (no possible poststate) and also nondeterministic programs (multiple possible poststates). We write

$$\omega[\![\alpha]\!]\nu$$

if the meaning of the program α relates prestate ω to poststate ν.

We define the meaning of assignment $x := e$ to evaluate e in the current state to c and then update the state to map x to c. In symbols:

$$\omega[\![x := e]\!]\nu \quad \text{iff} \quad \nu = \omega[x \mapsto c] \text{ where } c = \omega[\![e]\!]$$

Here we use the notation $\omega[x \mapsto c]$ for the result of updating the state ω by mapping x to c (no matter what it was before).

The meaning of sequential composition $\alpha \ ; \ \beta$ is to execute first α and then β from the resulting state. That is:

$$\omega[\![\alpha \ ; \ \beta]\!]\nu \quad \text{iff} \quad \text{there is a } \mu \text{ such that } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\nu$$

In other words, the relation denoted by $\alpha \ ; \ \beta$ is the *composition* of the relations denoted by α and β.

As an example, let's compute

$$(\omega[x \mapsto a])[\![x := x + 2]\!]\nu$$

and we find $\nu = (\omega[x \mapsto a])[x \mapsto a + 2] = \omega[x \mapsto a + 2]$ Slightly more complicated is

$$(\omega[x \mapsto a])[\![x := x + 1 \ ; \ x := x + 1]\!]\nu$$

We determine that there is an intermediate state $\mu = \omega[x \mapsto a + 1]$ and a final state $\nu = \omega[x \mapsto a + 2]$.

So, both of these programs define the same relation between $\omega[x \mapsto a]$ and $\omega[x \mapsto a + 2]$ Therefore we can state that these two programs are semantically equivalent

$$[\![x := x + 2]\!] = [\![x := x + 1 \ ; \ x := x + 1]\!]$$

They have the same meaning because they have the same effects on the state. This, by the way, might fail to be true if the language were extended to allow shared memory concurrency because another process can intervene after the first assignment on the right, while the left atomically increments x by two. Lesson: we always have to be careful about the extent of the language when we reason about it, be it semantically (as here) or logically (as in the next lecture).

4 Conditionals

We now add conditionals **if** $P\,\alpha\,\beta$ to our language, read as "*if* P *then* α *else* β". A characteristic of our setup is that formulas P do double duty: on one hand they serve as conditions in if-then-else programs and (shortly) guards on while loops. On the other hand we also use them to *reason* about programs as shown in the next lectures.

$$\begin{array}{lrcl} \text{Programs} & \alpha, \beta & ::= & x := e \mid \alpha \ ; \ \beta \mid \textbf{if}\ P\,\alpha\,\beta \mid \ldots \\ \text{Formulas} & P, Q & ::= & e_1 = e_2 \mid e_1 \leq e_2 \mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \\ & & \mid & P \to Q \mid \neg P \mid \ldots \end{array}$$

In concrete syntax we usually write $\top$ (top) as "**true**", $\bot$ (bottom) as "**false**", and $\neg$ as "**not**".

In order to define the meaning of the conditional, we first need to define the meaning of the formulas, in mathematical terms. Because variables range just over

integers, the language of formulas we are concerned with is that of *integer arithmetic*. We define their meaning relative to an assignment ω of values to variables

$$\omega \models P \qquad P \textit{ is true in state } \omega$$

It is defined on the structure of P.

$\omega \models \top$	always
$\omega \models \bot$	never
$\omega \models e_1 = e_2$	iff $\omega[\![e_1]\!] = \omega[\![e_2]\!]$
$\omega \models e_1 \leq e_2$	iff $\omega[\![e_1]\!] \leq \omega[\![e_2]\!]$
$\omega \models P \wedge Q$	iff $\omega \models P$ and $\omega \models Q$
$\omega \models P \vee Q$	iff $\omega \models P$ or $\omega \models Q$
$\omega \models \neg P$	iff $\omega \not\models P$
$\omega \models P \to Q$	iff whenever $\omega \models P$ then also $\omega \models Q$

If we extend the formula language with quantifiers, we add the following cases:

$$\begin{array}{lllll} \omega \models \forall x.\, P & \text{iff} & \omega[x \mapsto c] \models P & \text{for every } c \in \mathbb{Z} \\ \omega \models \exists x.\, P & \text{iff} & \omega[x \mapsto c] \models P & \text{for some } c \in \mathbb{Z} \end{array}$$

For the fragment of formulas above, the truth of a formula in a state is straightforward to decide by evaluating expressions in that state and then computing the Boolean connectives.

With this out of the way, we can now define the meaning of the conditional by cases on the truth of P.

$$\begin{array}{lll} \omega[\![\textbf{if } P\, \alpha\, \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\nu \text{ when } \omega \models P \\ & & \omega[\![\beta]\!]\nu \text{ when } \omega \not\models P \end{array}$$

5 While Loops

The abstract syntax for while loops is $\textbf{while}\, P\, \alpha$ which should somehow be the same as $\textbf{if}\, P\, (\alpha\ ;\ \textbf{while}\, P\, \alpha)\, \textbf{skip}$, where **skip** is a program that has no effect. Although it is perfectly possible to make this work as a so-called *inductive definition*, it has the issue that $\textbf{while}\, P\, \alpha$ appears on both sides. So we break it down by "guessing" the number of iterations of the loop, using an auxiliary relation $[\![\textbf{while}\, P\, \alpha]\!]^n$ indexed by an $n \geq 0$. If $n = 0$ we must exit the loop so P should be false, and if $n > 0$ we should go around the loop once, followed by $n - 1$ more iterations.

$$\begin{array}{lll} \omega[\![\textbf{while}\, P\, \alpha]\!]\nu & \text{iff} & \omega[\![\textbf{while}\, P\, \alpha]\!]^n\nu \\ & & \text{for some } n \geq 0 \\ \omega[\![\textbf{while}\, P\, \alpha]\!]^0\nu & \text{iff} & \omega \not\models P \text{ and } \omega = \nu \\ \omega[\![\textbf{while}\, P\, \alpha]\!]^{n+1}\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ for some } \mu \\ & & \text{and } \mu[\![\textbf{while}\, P\, \alpha]\!]^n\nu \end{array}$$

We can appeal to this definition to compute the meaning of a few simple programs. Actually, we will look at whole families of programs because it doesn't matter what some of the components are. For example, any program **while false** α will behave the same, regardless of α. Instead of looking up the answer immediately, we suggest solving these yourself first with careful reference to the definitions.

$$\begin{array}{l}\omega[\![\textbf{while true}\,\alpha]\!]\nu\\ \omega[\![\textbf{while false}\,\alpha]\!]\nu\\ \omega[\![x := x]\!]\nu\end{array}$$

We calculate

$$\begin{array}{ll}\omega[\![\textbf{while true}\,\alpha]\!]\nu & \text{never}\\ \omega[\![\textbf{while false}\,\alpha]\!]\nu & \text{iff } \nu = \omega\\ \omega[\![x := x]\!]\nu & \text{iff } \nu = \omega\end{array}$$

We see, for example, that

$$[\![\textbf{while false}\,\alpha]\!] = [\![x := x]\!]$$

where the equality here denotes an equality between two relations. Further examples in the next section.

6 Summary

$$\begin{array}{lcl}\omega[\![c]\!] & = & c\\ \omega[\![x]\!] & = & \omega(x)\\ \omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!]\\ \omega[\![e_1 * e_2]\!] & = & \omega[\![e_1]\!] \times \omega[\![e_2]\!]\end{array}$$

Figure 1: Semantics of Expressions

$$
\begin{array}{lll}
\omega[\![x := e]\!]\nu & \text{iff} & \omega[x \mapsto c] = \nu \text{ where } \omega[\![e]\!] = c \\
\omega[\![\alpha \,;\, \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\nu \text{ for some state } \mu \\
\omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\nu \quad \text{or} \\
 & & \omega \not\models P \text{ and } \omega[\![\beta]\!]\nu \\
\omega[\![\textbf{while } P \ \alpha]\!]\nu & \text{iff} & \omega[\![\textbf{while } P \ \alpha]\!]^n\nu \quad \text{for some } n \in \mathbb{N} \\
\omega[\![\textbf{while } P \ \alpha]\!]^{n+1}\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\textbf{while } P \ \alpha]\!]^n\nu \\
\omega[\![\textbf{while } P \ \alpha]\!]^0\nu & \text{iff} & \omega \not\models P \text{ and } \omega = \nu
\end{array}
$$

Figure 2: Semantics of Programs

$$
\begin{array}{lll}
\omega \models e_1 \leq e_2 & \text{iff} & \omega[\![e_1]\!] \leq \omega[\![e_2]\!] \\
\omega \models e_1 = e_2 & \text{iff} & \omega[\![e_1]\!] = \omega[\![e_2]\!] \\
\omega \models P \wedge Q & \text{iff} & \omega \models P \quad \text{and} \quad \omega \models Q \\
\omega \models P \vee Q & \text{iff} & \omega \models P \quad \text{or} \quad \omega \models Q \\
\omega \models P \rightarrow Q & \text{iff} & \omega \models P \quad \text{implies} \quad \omega \models Q \\
\omega \models \neg P & \text{iff} & \omega \not\models P \\
\omega \models P \leftrightarrow Q & \text{iff} & \omega \models P \quad \text{iff} \quad \omega \models Q \\
\omega \models \forall x.\, P & \text{iff} & \omega[x \mapsto c] \models P \quad \text{for every } c \in \mathbb{Z} \\
\omega \models \exists x.\, P & \text{iff} & \omega[x \mapsto c] \models P \quad \text{for some } c \in \mathbb{Z}
\end{array}
$$

Figure 3: Semantics of Formulas