Lecture Notes on Memory Safety

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

Lecture 6
January 29, 2026

1 Introduction

The classic *buffer overflow attack* (about which you learn in 15-213 *Computer Systems*) allows a program to take control of your machine while otherwise innocuous code is executed. It exploits that accessing memory that hasn't been explicitly allocated by a program is undefined behavior in C and therefore *unsafe*.

In this lecture we define a simplistic model of memory and introduce memory write and read operations into our language. We then define unsafe behavior and investigate how to prove safety of programs accessing memory. For the most part, we see that the reasoning principles involved are quite similar to how we have done normal assignment to variables, thanks to a theory of array-valued objects that allows us to treat memory updates as construction of new array-valued terms. As a result, the only explicit change in state that occurs is an update to whichever variable maps to the array.

2 A Proof of Safety

Before diving in to talk about the semantics of memory and memory safety, we start with a brief follow-up to our coverage of safety from the previous lecture. Proofs of safety can often be significantly simpler than proof of correctness. On the other hand, sometimes safety depends critically on some other correctness property.

We reconsider the example from Lecture 5.

$$x \geq 6 \rightarrow [\textbf{while } (x > 1)\ x := x - 2]\, 0 \leq x \leq 1$$

To prove this, we required a loop invariant, and $x \geq 0$ was sufficient.

We can modify this to introduce a division, and just prove safety (so the postcondition is $\top$.

$$x \geq 6 \land y = x \to [\textbf{while } (x > 2) \ \{x := x - 1 \ ; \ y := y - 1 \ ; \ z := \textbf{divide } z \ y\}] \top$$

After one step ($\to R$) it remains to prove

$$x \geq 6, y = x \vdash [\textbf{while } (x > 2) \ \{x := x - 1 \ ; \ y := y - 1 \ ; \ z := \textbf{divide } z \ y\}] \top$$

Here, we pick the weakest loop invariant we can think of, namely $J = (y = x)$. Then we have to prove:

True Initially: $x \geq 6, y = x \vdash y = x$, which is manifestly valid.

Preserved: We lose the antecedent $x \geq 6$ and $y = x$, but we add the loop invariant and the loop guard. So we have to show

$$y = x, x > 2 \vdash [\{x := x - 1 \ ; \ y := y - 1 \ ; \ z := \textbf{divide } z \ y\}](y = x)$$

Using the rules $[:=]R$ and $[\textbf{divide}]R$, this reduces to showing

$$y = x, x > 2, x' = x - 1, y' = y - 1 \vdash \neg(y' = 0)$$

and

$$y = x, x > 2, x' = x - 1, y' = y - 1, z' = z/y' \vdash y' = x'$$

Both of these are manifestly valid.

Implies Postcondition: Again, without the antecedent, but this time with the negated loop guard, we have to prove the postcondition $\top$. So:

$$y = x, \neg(x > 2) \vdash \top$$

Again, this is easily seen to be true.

So to prove safety, we need a nontrivial loop invariant in this example (which corresponds to proving a simple element of correctness as part of safety).

On the other hand, safety is still easy for the following modified program:

$$x \geq 6 \vdash [\textbf{while } (x > 1) \ \{y := \textbf{divide } y \ x \ ; \ x := x - 2\}] \top$$

By the loop guard we see that $x > 1$ inside the loop body, so the division is safe. However, if we had written $\textbf{divide } x \ y$ then there is an immediate counterexample with $y = 0$.

2.1 A Theorem about Safety[1]

Theorem 1 (Soundness of Dynamic Logic with Unsafe Programs) *All the rules of the sequent calculus are sound, and all the axioms we stated are valid.*

Proof: By considering each case and reasoning along similar lines as in the sample proofs of such properties in lecture. □

We can rigorously state that if we can prove *some* postcondition for α then α is safe. The theorem assumes that we have proved the soundness of all the sequent calculus rules (or axioms) we use in the formal proof (as claimed in the preceding theorem).

Theorem 2 (Safety) *If* $\cdot \vdash P \to [\alpha]Q$ *then there is no* ω *with* $\omega \models P$ *such that* $\omega[\![\alpha]\!]\lightning$.

Proof: Assume $\cdot \vdash P \to [\alpha]Q$ and for some ω we have $\omega \models P$ and $\omega[\![\alpha]\!]\lightning$. We have to show a contradiction.

By soundness of the sequent calculus we have $\omega \models P \to [\alpha]Q$. Since $\omega \models P$ we obtain $\omega \models [\alpha]Q$. By definition, this implies that **not** $\omega[\![\alpha]\!]\lightning$, which is a contradiction. □

3 Writing and Reading Memory

Mathematically, we model memory as a map from $\mathbb{Z}$ (the index domain) to $\mathbb{Z}$ (the value domain). It is total, but may be *indeterminate* on some indices. In the programming language we assume that each memory value comes with an allocated size, and accessing memory outside its bounds is *unsafe*.

As explained in Lecture 5, we would like to keep expressions *safe*, but possibly *indeterminate*. Unsafe behavior is then exhibited only by commands and the programs constructed from them. Sticking to this approach, we add a new kind of variable, M, that stands for memory (along with $N, \ldots$), and the new commands

- $M := \mathbf{alloc}\ e$ allocate a fresh memory block of size e and store it in M. This is unsafe if the value of e is negative.
- $M[e_1] := e_2$ write e_2 into memory M at address e_1. This is unsafe if the value of e_1 is out of bounds for M.
- $x := M[e]$ set x to the contents of memory at address e into x. This is unsafe if the value of e is out of bounds for M.

[1] mentioned, but not explicitly stated in lecture

The design decision that memory access takes place via commands means that you have to rewrite hypothetical code such as

$$M[x] := (M[x-1] + M[x+1])/2$$

in the more verbose form

$$\begin{array}{l} t_1 := M[x-1]\ ; \\ t_2 := M[x+1]\ ; \\ t_3 := \mathbf{divide}\ (t_1 + t_2)\ 2\ ; \\ M[x] := t_3 \end{array}$$

Next, we need to rigorously define the semantics of the new commands so that (a) we can implement them, and (b) we can prove soundness of our axioms and rules to reason about them (including their safety).

The first issue is how to track the contents of memory. For that purpose, we change our definition of the state ω. So far the state has been a total function from variables to integers, $\omega : \mathsf{Var} \to \mathbb{Z}$ where Var is the (countably infinite) set of variables. Now variables can also map to memory.

$$\omega : \mathsf{Var} \to (\mathbb{Z} \cup (\mathbb{Z} \to \mathbb{Z}))$$

We assume that programs map lowercase variables to integers and uppercase variables to memory, so that there is never any confusion between the two forms. Mathematically, we use the letter $H : \mathbb{Z} \to \mathbb{Z}$ (suggesting a *heap*). In addition, each heap comes with an allocated size $|H|$. All indices $0 \le i < |H|$ are in bounds; all other indices are out of bounds. All our previous semantic definitions remain unchanged since all variables in those definitions stand for integers.

We write $|H|$ for the allocated size of a heap H. This is a term and it is always determinate.

We also use $|M|$ as an expression. If $\omega[\![M]\!] = H$ then

$$\omega[\![|M|]\!] = |H|$$

We begin with safe and unsafe memory reads.

$$\begin{array}{rl} \omega[\![x := M[e]]\!]\nu & \text{iff} \quad \omega[\![e]\!] = i \text{ and } \omega[\![M]\!] = H \text{ and } \nu = \omega[x \mapsto H(i)] \\ & \qquad \text{provided } 0 \le i < |H| \end{array}$$

$$\omega[\![x := M[e]]\!]\lightning \quad \text{iff} \quad \omega[\![e]\!] = i \text{ and } \omega[\![M]\!] = H \text{ and } \textbf{not } 0 \le i < |H|$$

It should be clear that unsafe programs continue to satisfy no postcondition.

Memory write follows the same intuition, we just have to make sure to suitably update the map defining the state of memory.

$$\begin{array}{rl} \omega[\![M[e_1] := e_2]\!]\nu & \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \omega[\![e_2]\!] = a \text{ and } \omega[\![M]\!] = H \text{ and} \\ & \qquad H' = H[i \mapsto a] \text{ and } \nu = \omega[M \mapsto H'] \\ & \qquad \text{provided } 0 \le i < |H| \end{array}$$

$$\omega[\![M[e_1] := e_2]\!]\lightning \quad \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \omega[\![M]\!] = H \text{ and } \textbf{not } 0 \le i < |H|$$

Finally, allocation creates a fresh heap of the requested size. We do not model the initial contents, so the only information we obtain about the newly allocated heap H is its size.

$$\omega[\![M := \mathbf{alloc}\ e]\!]\nu \quad \text{iff} \quad \omega[\![e]\!] = n \text{ and } n \geq 0 \text{ and } \nu = \omega[M \mapsto H] \text{ for some } H \text{ with } |H| = n$$

$$\omega[\![M := \mathbf{alloc}\ e]\!]\lightning \quad \text{iff} \quad \omega[\![e]\!] = n \text{ and } n < 0$$

Before moving on, it is worth taking note that we explicitly do not allow assigning one heap variable to another, i.e., we do not consider $N := M$ to be a valid program. If we did, we would need to model this semantically as copying the contents of M into a new heap N. This can be accomplished by iterating over the length of M and copying each memory cell into N.

4 Reasoning about Memory

In order to reason about memory we need to introduce expressions that capture what we know about the state of memory. Mathematically, this leads us to the *theory of arrays* [**?**] which we can view as being constructed on top of the theory of arithmetic we have assumed so far. Because of the importance of arrays in imperative programming, some efficient decision procedures have been devised (see, for example, **?**) and implemented in provers such as Z3 or the CVC family.

In the theory of arrays we have two expressions $\mathbf{read}\ H\ i$ and $\mathbf{write}\ H\ i\ a$ where H denotes an array, i and index into an array, and a a value stored in the array. Note that the expression $\mathbf{write}\ H\ i\ a$ denotes a "new" array; semantically $H[i \mapsto a]$. For us, both the index domain and the values are integers.

To reason about safety of memory access we also need to talk about the allocated size of a heap. Mathematically, we write $|H|$ for this size. Writes preserve the allocated size, so we have the axiom

$$|\mathbf{write}\ H\ i\ a| = |H|$$

There are two axioms, called *read over write*, that allow us to reason about these expressions.

$$i = k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = a$$
$$i \neq k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = \mathbf{read}\ H\ k$$

In addition we have an *axiom of extensionality* which states that two arrays are equal if they agree on all elements. In our use of the theory of arrays, the quantifier ranges over integers.

$$(\forall i. H(i) = H'(i)) \wedge |H| = |H'| \rightarrow H = H'$$

As usual, we treat the new expressions as always denoting either an array or an integer, although the value may sometimes be indeterminate.

The right rules of the sequent calculus for these new commands are now relatively straightforward.

$$\frac{\Gamma \vdash 0 \leq e < |M|, \Delta \quad \Gamma, x' = \mathbf{read}\ M\ e \vdash Q(x'), \Delta}{\Gamma \vdash [x := M[e]]Q(x), \Delta}\ [\mathbf{read}]R^{x'}$$

As for assignment, the x' must be chosen fresh in $[\mathbf{read}]R^{x'}$. The same is true for M' in the following rules.

$$\frac{\Gamma \vdash 0 \leq e_1 < |M|, \Delta \quad \Gamma, M' = \mathbf{write}\ M\ e_1\ e_2 \vdash Q(M'), \Delta}{\Gamma \vdash [M[e_1] := e_2]Q(M), \Delta}\ [\mathbf{write}]R^{M'}$$

For allocation, the main thing that we need to be careful about is breaking any association with the newly-allocated heap and the previous heap's contents. This is accomplished by creating a fresh variable M' and introducing a single assumption about its size.

$$\frac{\Gamma \vdash 0 \leq e, \Delta \quad \Gamma, |M'| = e \vdash Q(M'), \Delta}{\Gamma \vdash [M := \mathbf{alloc}\ e]Q(M), \Delta}\ [\mathbf{alloc}]R^{M'}$$

The aspect of these rules critical for safety is the first premise that requires us to prove safety (independently of the postcondition).

4.1 Uninitialized Memory

A student in class asked what happens if we apply these rules in a situation where an array is freshly allocated, and then read from prior to any writes taking place. This is a good opportunity to see them in action.

Supposing we have a program $M := \mathbf{alloc}\ 5; x := M[0]$, and we want to query whether $0 < x$ afterwards. We start by setting this up in a sequent, and removing the sequential composition.

$$\frac{\dfrac{\cdot \vdash 0 \leq 5 \quad |M'| = 5 \vdash [x := M'[0]]0 < x}{\cdot \vdash [M := \mathbf{alloc}\ 5][x := M[0]]0 < x}\ [\mathbf{alloc}]R^{M'}}{\cdot \vdash [M := \mathbf{alloc}\ 5; x := M[0]]0 < x}\ [;]R$$

The left premise is true by arithmetic. Continuing with the right premise,

$$\frac{|M'| = 5 \vdash 0 \leq 0 < |M'| \quad |M'| = 5, x' = \mathbf{read}\ M'\ 0 \vdash 0 < x'}{|M'| = 5 \vdash [x := M'[0]]0 < x}\ [\mathbf{read}]R^{x'}$$

Again the left premise follows immediately by arithmetic, but the right premise is stuck. This answers the original question: just like with uninitialized variables, we simply can't prove anything about the uninitialized contents of memory. Any attempt to prove non-vacuous properties involving uninitialized memory will at some point get stuck as we see in this example.

5 A Small Example of Memory Safety

Consider the following program to initialize memory up to n:

$$i := 0 \ ; \ \textbf{while} \ (i < n)\{ \ M[i] := i \ ; \ i := i + 1 \ \}$$

This is patently unsafe: just consider a state where $|M| = k$ and $n = k+1$. Then the last time around the loop we will have $i = k$, leading to an unsafe memory access at $M[k]$.

We can add a precondition $n \leq |M|$ and then try to prove safety with

$$n \leq |M| \rightarrow [i := 0 \ ; \ \textbf{while} \ (i < n)\{ \ M[i] := i \ ; \ i := i + 1 \ \}]\top$$

We could try $i \leq n$ but that's insufficient. Here is what preservation would require:

$$i \leq n, i < n \vdash [M[i] := i \ ; \ i := i + 1]i \leq n$$

We note two problems: (1) safety will fail because we cannot prove that $0 \leq i$ and (2) we have lost the precondition $n \leq |M|$ so we also cannot conclude that $i < |M|$.

Let's try a more complex invariant:

$$0 \leq i \leq n \leq |M|$$

Now preservation requires

$$0 \leq i \leq n \leq |M|, i < n \vdash [M[i] := i \ ; \ i := i + 1] \, (0 \leq i \leq n \leq |M|)$$

This reduces in two steps to

$$0 \leq i \leq n \leq |M|, i < n, M' = \textbf{write} \ M \ i \ i, i' = i + 1 \vdash 0 \leq i' \leq n \leq |M'|$$

Fortunately, this is valid (even with the useless assumption about M'). Because our postcondition is just $\top$, that is easily seen to be true, but the loop invariant does not hold initially because

$$n \leq |M| \vdash 0 \leq 0 \leq n \leq |M|$$

is not valid (counterexample: $n = -1$). So we need to strengthen our precondition to

$$0 \leq n \leq |M|$$

6 Summary

Since it has been a while, we summarize the language so far. We restrict programs from containing certain expressions with indeterminate behavior to retain

the property that for every given prestate ω, every program has three possible outcomes: a poststate ν, or no poststate in which case it may be safe or unsafe ($\lightning$).

Variables	x, y, z			
Memory	$M, N, \ldots$			
Constants	c	$::=$	$\ldots, -1, 0, 1, \ldots$	
Expressions	e	$::=$	$c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$	determinate
		$\mid$	$\lvert M \rvert$	determinate
		$\mid$	$e_1/e_2 \mid \textbf{read}\ M\ e \mid \textbf{write}\ M\ e_1\ e_2$	may be indeterminate
Programs	α, β	$::=$	$x := e \mid \alpha\ ;\ \beta \mid \textbf{skip}$	
		$\mid$	$\textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta \mid \textbf{while}\ P\ \alpha$	
		$\mid$	$\textbf{test}\ P$	safe
		$\mid$	$\textbf{assert}\ P \mid x := \textbf{divide}\ e_1\ e_2$	may be unsafe
		$\mid$	$M := \textbf{alloc}\ e$	may be unsafe
		$\mid$	$x := M[e] \mid M[e_1] := e_2$	may be unsafe
Formulas	P, Q	$::=$	$e_1 \leq e_2 \mid e_1 = e_2 \mid \top \mid \bot$	
		$\mid$	$P \wedge Q \mid P \vee Q \mid P \to Q \mid P \leftrightarrow Q \mid \neg P$	
		$\mid$	$\forall x.\, P(x) \mid \exists.\, P(x)$	
		$\mid$	$[\alpha]Q \mid \langle\alpha\rangle Q$	