Software Security

'Safe' programming languages (part 2) and TOCTOU/race conditions

Erik Poll

Radboud Universiteit Nijmegen

Common combinations of memory & type safety

- memory-unsafe, typed, but type-unsafe: C, C++, assembly

  Pointer arithmetic or unsafe type constructions you can break any guarantees the type system tries to make - examples later
- memory-safe, typed, and type-safe:
  - Java, C#, Rust, Go

    Possibly with some loopholes to allow unsafety
  - Functional languages such as Haskell, F#, ML, Clean
  - Some *(dynamically)* typed languages, eg JavaScript, still allow pretty weird behaviour
- memory-safe and untyped (unusual): LISP, Prolog

Memory-safety and type-safety are mutually dependent:

- memory corruption can break type guarantees
- lack of type safety can break memory safety guarantees

Recap: Memory unsafety

A programming language is memory-safe if it guarantees that

1. programs never access unallocated or de-allocated memory
2. possibly also initialization safety
   program never access *uninitialised* memory

Unsafe language features that break memory safety:

1. no array bounds checks
2. pointer arithmetic
3. null pointers, *but only if these cause undefined behaviour*
4. manual memory management with eg. malloc, new & free
   Avoiding this requires automated memory management
   or not using the heap
5. some forms of type unsafety

Automated memory management

- Tracing garbage collector in LISP, Java, C#, …
  gc process periodically traces which objects are reachable from the root objects; unreachable objects are de-allocated; most common solution

  + invisible to the programmer
  -- periodic unpredictable overhead; some variants make it predictable

- Automatic Reference Counting (ARC) in Objective C, Swift & Apple OSs
  Compiler adds code to keep track of how many references there are to an object; when count reaches zero objects are deallocated

  + No unpredictable delays
  -- Not invisible to programmer esp. for cyclical data structures

- Ownership approach in Rust
  Compiler inserts free() statements for objects in the right places, namely when the owner can be freed()

  + No unpredictable delays
  -- Not invisble to the programmer
  they need to be aware of ownership;
  as with ARC, cyclical data structures need special attention

Memory unsafety in safe languages?

Memory corruption can still happen is memory safe languages like Java, C# or Rust

1. in native code
   (i.e. unsafe binary code called from safe language)
2. for C# and Rust, in code blocks declared as `unsafe`
3. through bugs in the Virtual Machine (VM) implementation, which is typically written in C++....
4. through bugs in the implementation of the type checker
   or – worse – through bugs in the type system (unsoundness)

3 & 4 can be understood by thinking about the Trusted Computing Base (TCB) for memory-safety and type-safety:

The VM (incl. type checker aka byte code verifier) is part of this TCB; so bugs in it can break these properties.

Type-safety

Guarantees that programming language can provide

```
public class Demo{
  static private string greeting = "Hello";
  final static int CONST = 42;

  static void  Main (string[]  args){
     foreach (string name in args){
         Console.Writeline(sayHello(name));
     }
  }

  public static string sayHello(string name){
     return greeting + name;
  }
}
```

`greeting` only accessible in class Demo

`CONST` will *always* be 42

sayHello will always return a string

sayHello will always be called with 1 parameter of type string

Types

- Types assert invariant properties of program elements. Eg
  - This variable will always hold an integer
  - This function will always return an object of class X (or one of its subclasses)

- Type checking verifies these assertions. This can be done
  - at compile time (static typing) or
  - at runtime (dynamic typing)

  or a combination.

Run & compile time checks in Java or C#

Ruled out at language-level, by combination of

- compile-time typechecking (static checks)
  - or at load-time, by bytecode verifier (bcv)
- runtime checks (dynamic checks)

What runtime checks are performed when executing the code below?

```
public  class A extends Super{
  protected int[] d;
  private A next;

  public A() { d = new int[3]; }
  public void  m(int j) { d[0] = j; }
  public setNext(Object s)
        next = (A)s;
  }
}
```

Runtime checks for 1) non-nullness of d and 2) array bound.

runtime check for type (down)cast

Run & compile time checks in Java or C#

Can we omit the array bound check in `m(int j)` ?

Given that `d` has length 3 ?

No, because a subclass of A can change the value of d

```
public  class A extends Super{
   protected int[] d;
   private A next;

   public A() { d = new int[3]; }
   public void  m(int j) { d[0] = j; }
   public setNext(Object s)
         next = (A)s;
   }
 }
```

Runtime check
if 0 is within
array bounds

Type-safety

Type safety (aka type soundness or strong typing)

A language is type safe if the type assertions are guaranteed to hold at run-time

Type-safety guarantees that
programs can only manipulate data in ways allowed by their types

or at least for programs that pass the type-checker

- So you cannot multiply booleans, dereference an integer, take the square root of reference, etc.

  Note: this removes lots of room for UNDEFINED BEHAVIOUR

- For OO languages: no "Method not found" errors at runtime

Breaking type safety with unsafe union types

```
struct user{
     bool is_enrolled;
     bool has_paid_tuition_fee;
     union {
       char* name; // for users that have no uid yet
                   //  "OR"
       int uid;    // for users that have a uid
      };
   };
...
struct user u;
u.is_enrolled = true; u.uid = 250012;
printf("Username is %s \n", u.name);
```

- Unions aka variants in C/C++ and some other languages are type unsafe
  Behaviour of code above is unpredictable & depends on underlying data representations; it may allow pointer arithmetic
- There are also type-safe union/variant constructs;
  then accessing a union requires code for all cases.
  Eg the class template `std::variant` in C++ is type-safe

Breaking type safety with unsafe casts

```
class DiskQuota {
  private:
     int MinBytes;
     int MaxBytes;
};

void EvilCode(DiskQuota* quota) {
   // use pointer arithmetic to access
   // the quota object in any way we like!
   ((int*)quota)[1] = MAX_INT;

}
```

For a C(++) program we can make *no guarantees whatsoever* in the presence of untrusted code.

So

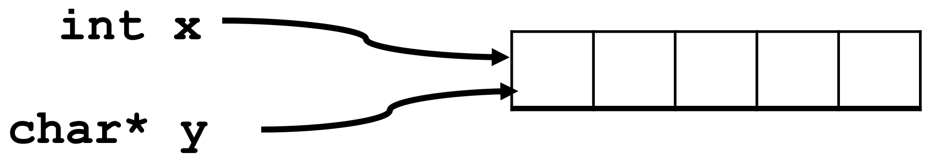
- a buffer overflow in some library can be fatal
- in a code review we have to look at *all code* to make guarantees

Breaking type safety?

Type safety is an extremely fragile property:

one tiny flaw brings the whole type system crashing down

Data values and objects are just blobs of memory. If we can create type confusion, by having two references with different types pointing the same blob of memory, then *all* type guarantees are gone.



- Example: type confusion attack on Java in Netscape 3.0:

```
public class A[]{ ... }
```

Netscape's Java execution engine confused this type `A[]` with the type array of `A`

Root cause: [and] should not be allowed in class names

So this is an input validation problem!

Type confusion attacks

```
public class A{
    public Object x;
     ...
}
```

What if we could compile B against A but we run it against A?

We can do pointer arithmetic again!

If Java Virtual Machine would allow such so-called *binary incompatible* classes to be loaded, the whole type system would break.

```
public class A{
  public int x;
   ...
}

public class B{
 void setX(A a) {
  a.x = 12;
 }
}
```

Representation independence

- Can we tell if `true` is represented as `0x00` and `false` as `0xFF` or vv?

- Can we tell is `Strings` are represented as null-terminated character array

h	e	l	l	o	\0

or as records with a length field?

5	h	e	l	l	o

If (well-typed) programs can determine how data structures are represented then we have representation independence

So representation independence => type safety

Trends

Trends memory safety (not exam material)

Clear trend towards more memory safety

- Several large industry players joined Rust foundation

  aws Google HUAWEI Meta Microsoft

  Some new Linux, Chrome and Android code now in Rust

  Unfortunately, CISA's efforts to push memory safety have died down

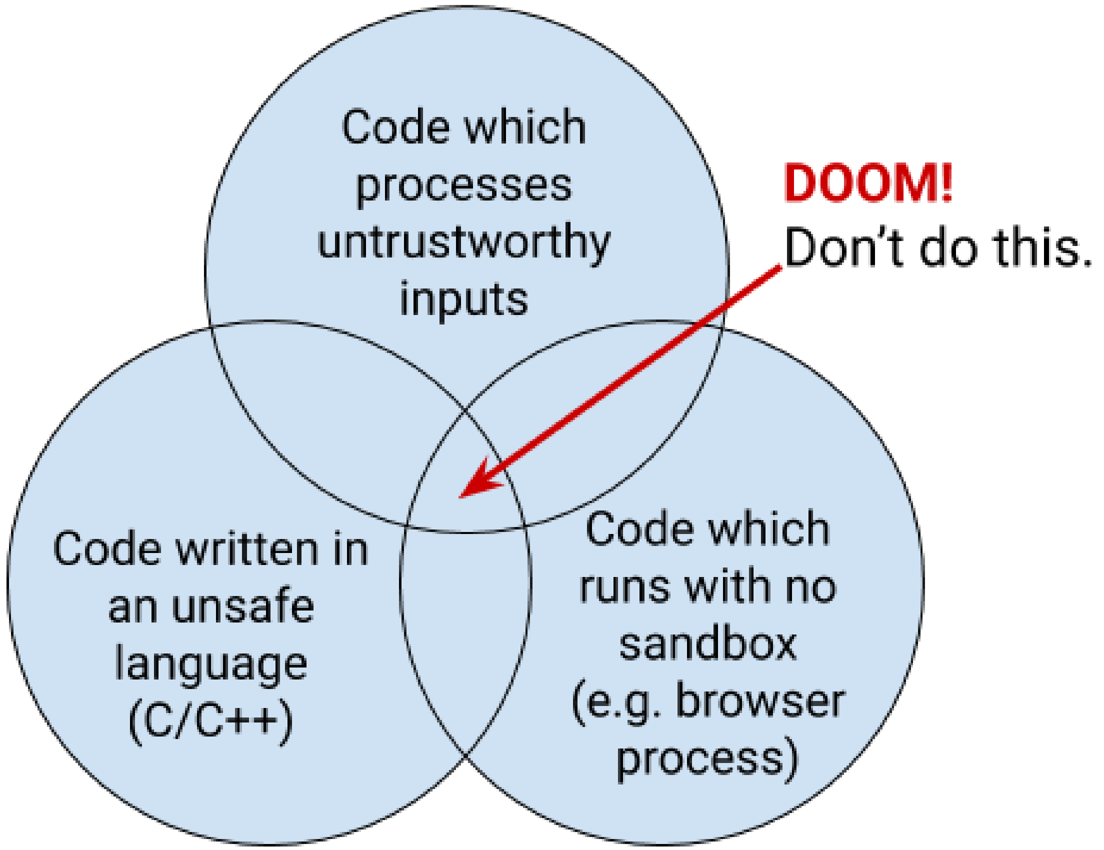
- Apple has started to leave runtime checks for bounds safety in production code, to prevent (some) spatial bugs, but not temporal bugs

  [Yeoaul Na's keynote talk at LLVM'23
  "-fbounds-safety": Enforcing bounds safety for production C code
  https://www.youtube.com/watch?v=RK9bfrsMdAM]

  Sept 2025 Apple announced an initiative for improve up memory tagging to also catch temporal bugs

  https://security.apple.com/blog/memory-integrity-enforcement/

Chromium Rule of Two


Code which processes untrustworthy inputs
DOOM!
Don't do this.
Code written in an unsafe language (C/C++)
Code which runs with no sandbox (e.g. browser process)

Trends in type safety

Trend towards more type safety & more expressive type systems. Eg.

- TypeScript replacing JavaScript (more on that in later lecture)
- Programming languages distinguishing non-null types vs nullable types (aka option(al) types)

```
public @NonNull String hello = "hello";
```

to catch null pointer bugs earlier, at compile time (and maybe also improve efficiency)

- – C# supports nullable types, written as `A?` or `Nullable<A>`
- – In Java you can use type annotations `@Nullable` and `@NonNull`
- – Scala, Rust, Kotlin, Swift, and Ceylon have non-null types vs nullable types

Languages then usually adopt the approach that references are non-null by default (as PREfast did)

Other forms of safety

Safe arithmetic

What happens if `i=i+1;` overflows?

What would be unsafe or safe(r) approaches?

1. *Unsafest approach* : leaving this as undefined behavior
   - eg C and C++
2. *Safer approach* : specifying how over/underflow behaves
   - eg based on 32 or 64 bit two-complements behaviour
   - eg Java and C#
3. *Safer still* : integer overflow results in an exception
   - eg checked mode in C#
4. *Safest*: have infinite precision integers & reals, so overflow never happens

   Python and functional programming languages like Haskell have infinite precision integers.
   There have been experiments with infinite precision reals, but no mainstream programming languages provide these AFAIK.

Visibility

Visibility modifiers: `public` , `private` , ...

allow encapsulation aka compartmentalization

that can provide safety guarantees, eg. invariants

```
public class A {
    private int i = 1;   // invariant: i will always be positive

    public set_i (int j) {
        if (j > 0) { i = j;}
    }

    public int get_i ()  {
        return i;
    }
   ...
}
```

Language level sandboxing

Java and C# / .NET originally provided sandboxing,

aka code-based access control

where eg different classes could be given different permission

Eg permissions to access the network,
or permissions to access certain files or directories, …

with clever mechanism (stackwalking) to prevent 'trusted' classes with high privileges to be abused by 'untrusted' classes with low priviliges

Immutability

- of primitive values (ie constants)
  - in Java : `final int i = 5;`
  - in C(++) : `const int BUF_SIZE = 128;`

  Beware: meaning of `const` is confusing for C(++) pointers & objects!

- of objects
  - In Java, for example `String` objects are immutable
    - (immutable) strings can be implements using (mutable) arrays, thanks to visibility modifiers

Scala, Rust, Ceylon, and Kotlin provides a more systematic distinction between mutable and immutable data to promote the use of immutable data structures

In functional programming languages data structures are always immutable, so we get this for free!

AND NOW FOR SOMETHING COMPLETELY DIFFERENT

Spot the security flaws

```
public class SimpleServlet extends HttpServlet {
  private String query;
  public void doGet(HttpServletRequest request,
                              HttpServletResponse response)
                           throws ServletException, IOException {
    try { Connection conn =
                DriverManager.getConnection("jdbc:odbc ... ");
         query = "INSERT INTO roles" + "(userId, userRole)" +    "VALUES "  + "('"  +
                    request.getParameter("userId") + "'," + "'standard')";
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(query);
        } catch ...
  }
```

Spot the security flaws

```
public class SimpleServlet extends HttpServlet {
  private String query;
  public void doGet(HttpServletRequest request,
                              HttpServletResponse response)
                           throws ServletException, IOException {
    try { Connection conn =
               DriverManager.getConnection("jdbc:odbc ... ");
        query = "INSERT INTO roles" + "(userId, userRole)" +   "VALUES "  + "('"  +
                   request.getParameter("userId") + "','" + "'standard')";
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(query);
        } catch ...
  }
```

Concurrent calls of doGet will act on the *same* Servlet object and hence use the *same* instance field `query`

SQL injection

Spot the security flaws

```
public class SimpleServlet extends HttpServlet {
  ~~private String query;~~
  public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
                       throws ServletException, IOException {
   String query;
    try { Connection conn =
               DriverManager.getConnection("jdbc:odbc ... ");
         query = "INSERT INTO roles" + "(userId, userRole)" +    "VALUES "  + "('"  +
                    request.getParameter("userId") + "','" + "'standard')";
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(query);
        } catch ...
  }
```

Fix: now every (possibly concurrent) call of doGet has its own `query` field

Thread-safety

Problems with threads (ie. lack of thread safety)

- Two concurrent execution threads both execute the statement

  ```
  x = x+1;
  ```

  where `x` initially has the value 0.

  What is the value of `x` *in the end?*

  Answer: `x` can have value 2 or 1
  In some languages `x` can have any value

- The root cause of the problem is a data race:
  `x = x+1` is *not* an atomic operation, but happens in two steps - reading `x` and assigning it the new value - which may be interleaved in unexpected ways

- Why can this lead to security problems?

  Think of internet banking, and running two simultaneous sessions with the same bank account… *Do try this at home!* ☺

Data races and thread-safety

- A program contains a data race if two execution threads simultaneously access the same variable and at least one of these accesses is a write

  NB data races are highly non-deterministic, and a pain to debug!

- Thread-safety = the behaviour of a program consisting of several threads can be understood as an interleaving of those threads
  - Rust is thread-safe

    (but it does not ensure deadlock-freeness)
  - Nearly all other imperative languages, incl. C, C++, Java, C#, are not thread-safe; it is up to the programmer to ensure thread-safety.
  - In Java and C# data races can result in arbitrary values, but won't break memory- or type-safety
  - Some programming languages introduce features to help with thread safety, e.g. `@ThreadLocal` annotations in Kotlin

Example: Threads causing problems...

```
public class A {
   private int i = 1;   // invariant: i will always be positive

   public set_i (int j) {
      if (j > 0) { i = j;} ;
   }

   public int get_i ()  {
      return i;
  }
  ...
}
```

I lied to you earlier.

The invariant for this class can be broken in a multi-threaded Java program

How?

- Two threads invoking set_i() at the same time could result in an arbitrary value for i
- Also, a thread could observe the value 0 for i for an A object while another thread is busy initialising that object

Example: Threads causing really unexpected problems

```
class A {
        private int i ;
        A() { i = 5 ;}
        int get_i() { return i; }
}
```

Can geti() ever return something else than 5?

Yes!

Thread 1, initialising x

```
    static A x = new A();
```

Thread 2, accessing x

```
    j = x.get_i();
```

You'd think that here x.geti() returns 5 or throws an exception, depending on whether thread 1 has initialised x

Hence: x.get_i() in thread 2 can return 0 instead of 5

Execution of thread 1 takes in 3 steps

1. allocate new object m
2. m.i = 5;
3. x = m;

the compiler or VM is allowed to swap the order of these statements, because they don't affect each other

Example: fixing this in Java

```
class A {
    private final int i ;
    A() { i = 5 ;}
    int geti() { return i;}
}
```

Now geti() always return 5.

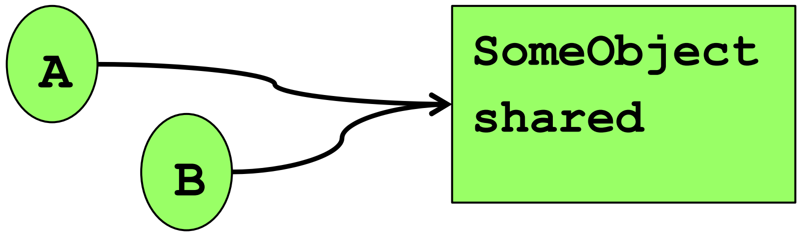
Declaring a private field as final fixes this particular problem

- This is a totally ad-hoc fix; the JVM spec includes some ad-hoc restrictions on the initialisation of `final` fields
- The API implementation of String was only fixed in Java 2 (aka 1.5); prior to this, strings were not really immutable in multi-threaded code

Dangers of aliasing

Dangerous combination: **ALIASING** & **MUTATION**

Aliasing: two threads or objects A and B both have a reference to the same object `shared`



This is the root cause of many problems, not just with concurrency

1. in concurrent (aka multi-threaded) context: data races
   – Locking objects (eg `synchronized` methods in Java) can help, but: expensive & risk of deadlock
2. in single-threaded context: dangling pointers
   – Who is responsible for free-ing `shared`? A or B?
3. in single-threaded context: broken assumptions
   – If `A` changes the `shared` object, this may break `B`'s assumptions about this object

References to mutable data are dangerous

In multi-threaded programs, aliasing of mutable data structures can be problematic, as the referenced data can change,

even in safe programming languages such as Java or C# !

```
public void f(char[] x){
  if (x[0] != 'a') { throw new Exception(); }
  // Can we assume that x[0] is the letter 'a'  here?
  // No!! Another concurrent execution thread could
  //      change the content of x at any moment
```

If there is aliasing, other threads can modify the content of x at any moment.

- Other threads could do this accidentally or deliberately
- Especially a concern if there are trust boundaries between more trusted and less trusted pieces of code

References to *im*mutable data are *less* dangerous

In a multi-threaded program, aliasing of immutable data structures are safer.

```
public void f(String x){
 if (x.charAt(0) != 'a') { throw new Exception(); }
 // We CAN assume that x[0] is the letter 'a` here?
 // Yes, as Java Strings are immutable
   ...
```

Another thread with a reference to the same string *cannot* change the value (or ‘contents’) of the string, as Java strings are immutable.

Kotlin has annotation `@SharedImmutable` to explicitly mark objects as being immutable & (therefore) safe to share

Concurrency & security

Spot the security flaw

Code inside an operating system, or some other system that performs access control

```
const char *filename = "/tmp/erik";
 if (access(filename, R_OK)!=0) {
     ... // handle error and exit;
 }
 // file exists and we have access
 int fd open(filename, O_RDONLY);
 ...
```

Between calls to `access` and `open` the file might be removed, or a symbolic link in the path might be reset!

TOCTOU (Time of Check, Time of Use)
aka
non-atomic check and use

Data races & TOCTOU

- Race condition aka data race is a common type of bug in concurrent programs
  - Basically: two execution threads mess with the same data or object (program variable, file, ...) at the same time
  - Not necessarily a *security* bug, but it can be...

- TOCTOU (Time Of Check, Time of Use)

  aka Non-atomic check and use

  is a closely related security flaw:

  Some precondition required for an action is invalidated between the time it is checked and the time the action is performed

  - Typically, this precondition is access control condition
  - Typically, it involves some concurrency

Race condition & file systems

Interaction with the file system is common source of TOCTOU issues

Signs of trouble:

- Access to files using filenames rather than file handles or file descriptors
  - filenames may point to different files at different moments in time
- Creating files or directories in publicly accessible places, for instance `/tmp`
  - especially if these have predictable file names

Classic UNIX security flaw

`lpr -r`

- Print utility with `-r` option to remove file after printing
- Could be used to delete arbitrary files

How?

1. User gives the command `lpr -r symlink` where `symlink` is a symbolic link
2. OS checks that user has permission to read & delete this file
3. While the file is printing move the link is moved, eg to `/etc/passwd`
4. after printing `lpr`, which has *root permission*, deletes `/etc/passwd`

Root of the problem: time between check (2) and use (4)

People keep making the same mistake

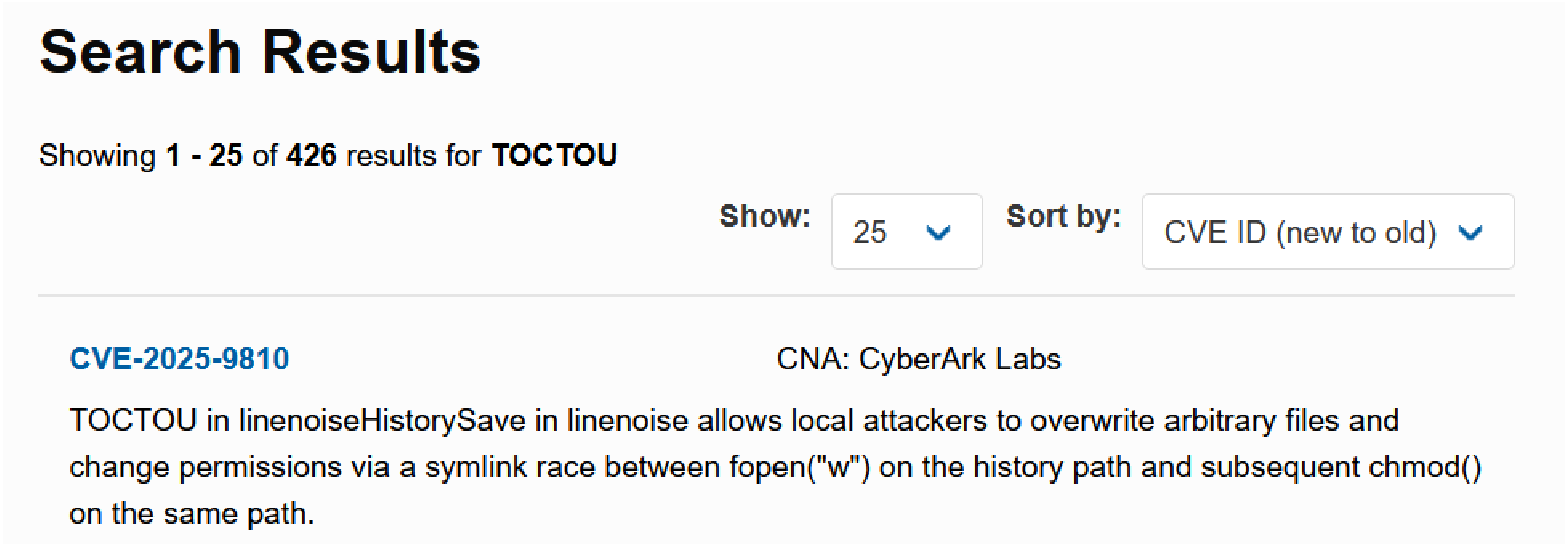
`lpr -r` is a classic security flaw from the 1970s

but similar flaws happened decades later

> CVE-2003-1073
> A race condition in the at command for Solaris 2.6 through 9 allows local users to delete arbitrary files via the -r argument with .. sequences in the job name, then modifying the directory structure after at checks permissions to delete the file and before the deletion actually takes place

Combination of race condition with failure to check that file names do not contain ..

People really keep making the same mistake



Search Results

Showing **1 - 25** of **426** results for **TOCTOU**

CVE-2025-9810 CNA: CyberArk Labs

TOCTOU in linenoiseHistorySave in linenoise allows local attackers to overwrite arbitrary files and change permissions via a symlink race between fopen("w") on the history path and subsequent chmod() on the same path.

Another classic: `mkdir` on Unix

- `mkdir` creates a new directory/folder
- This program executes as root
    - in Linux terminology, it is `setuid root`
- It creates new directory *non-atomically*, in several steps:
    1. enter super-user mode
    2. creates the directory, with owner is root
    3. sets the owner, to whoever invoked `mkdir`
    4. exit super-user mode
- Attack: by creating a symbolic link between steps 2 and 3, attacker can own any file

Two kinds of data races

There are two kinds of data race problems

1) *within* a (multi-threaded) program

    Eg the example with shared instance variable `query`

2) *in interaction* of a (possibly single-thread ed) program with the *outside world*

    Eg the examples involving the file system

1) can lead to accidental bugs that an attacker might exploit
2) allows attacker to cause and control the data race

Malicious code could deliberately introduce problems of type 1

Two attacker models

- **The I/O attacker**

  The I/O attacker can feed malicious inputs and observe outputs.
  Here the attacker can exploit bugs that are in the code accidentally

  This is the normal attacker model for software.

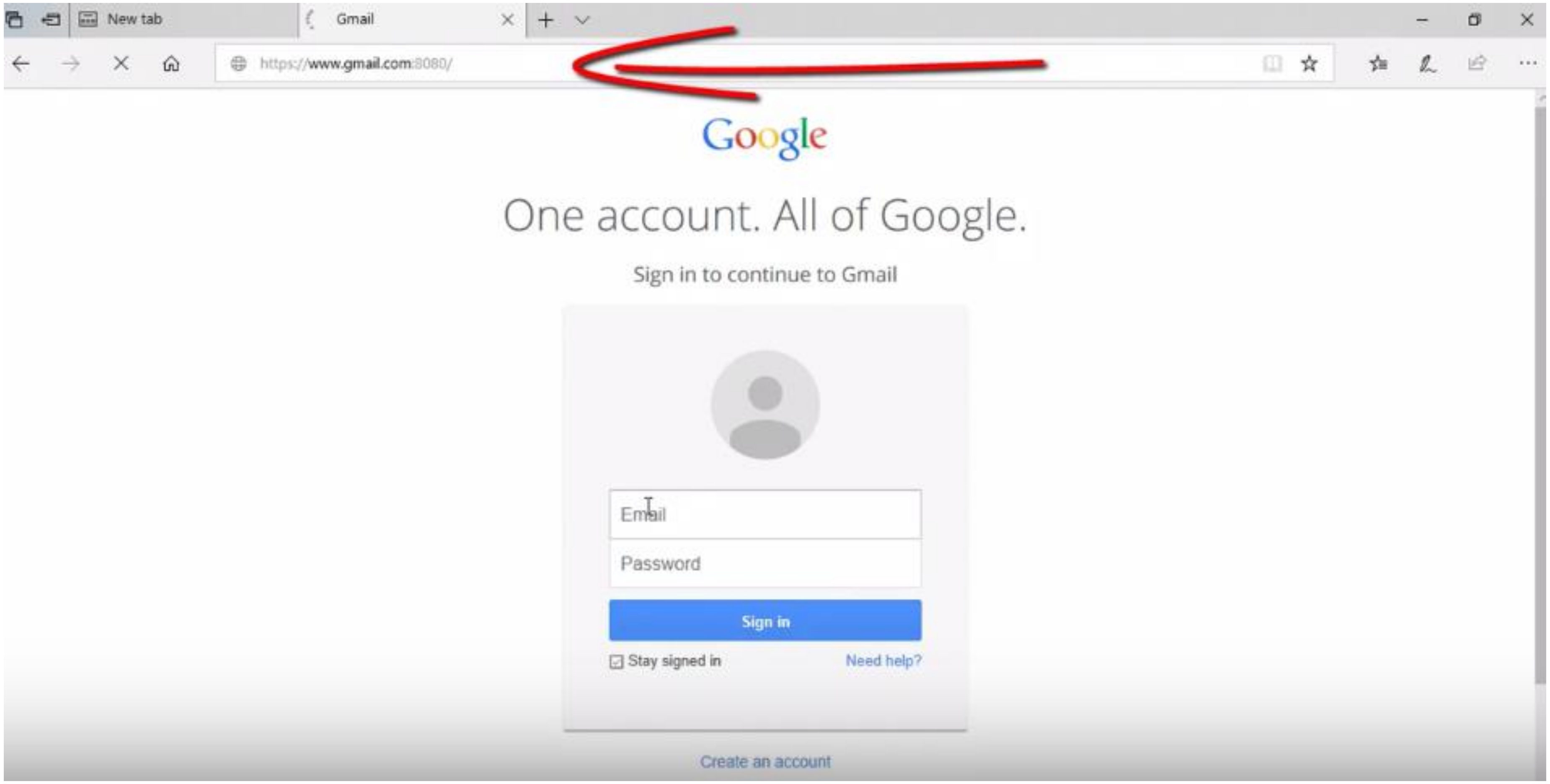
- **The malicious code attacker**

  More powerful attacker, who can introduce malicious code.

  How?

  Malicious insider or supply chain attack

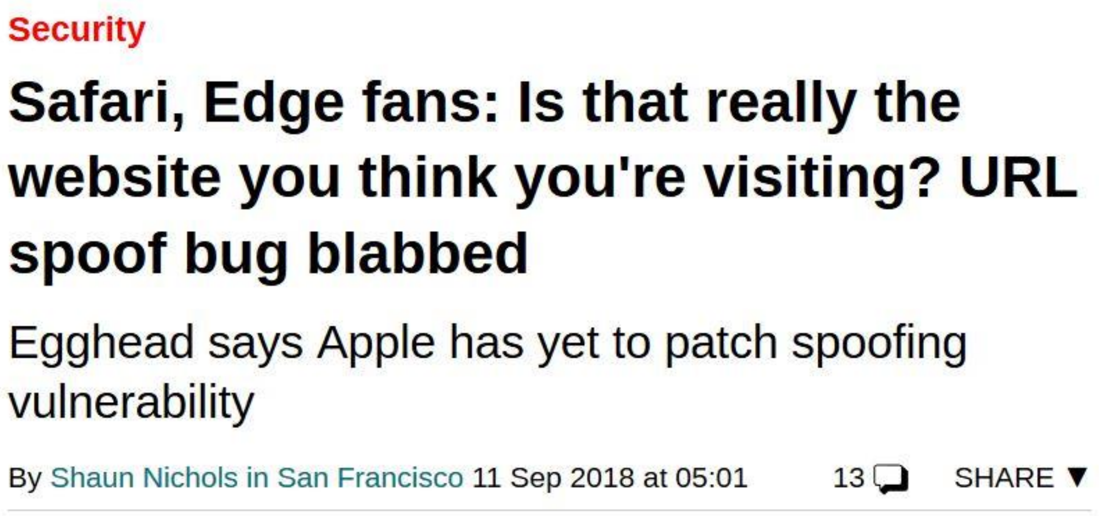
Is it safe to enter your credentials?



One account. All of Google.

Sign in to continue to Gmail

Edge & Safari GUI bug [CVE-2018-8383]


Security

Safari, Edge fans: Is that really the website you think you're visiting? URL spoof bug blabbed

Egghead says Apple has yet to patch spoofing vulnerability

By Shaun Nichols in San Francisco 11 Sep 2018 at 05:01 13 SHARE ▼

URL in address bar could be spoofed with a race condition:

JavaScript code loads legitimate page;
changes address bar, but over non-existent port;
and then quickly loads another page

https://www.theregister.co.uk/2018/09/11/safari_edge_spoofing/

https://youtu.be/Ni2XzF5-ixY

https://youtu.be/dGJSsK55nfQ