

Using Type Systems to Reduce your Security Risk

Zax
Rutgers University, NJ

October 21, 2007

The Problem

- User input is poisonous

The Problem

- User input is poisonous
- Unintended interactions in the program happen

The Problem

- User input is poisonous
- Unintended interactions in the program happen
- Data gets interpreted as code

The Problem

- User input is poisonous
- Unintended interactions in the program happen
- Data gets interpreted as code
- The program's behavior is now under user control

The Problem

- User input is poisonous
- Unintended interactions in the program happen
- Data gets interpreted as code
- The program's behavior is now under user control
- The program provides a portal to the system

The Problem

- User input is poisonous
- Unintended interactions in the program happen
- Data gets interpreted as code
- The program's behavior is now under user control
- The program provides a portal to the system
- Rinse and Repeat

Sql Example

Stopping this is trickier than it seems

Sql Example

Stopping this is trickier than it seems

Let's write some exciting database code to see why

Sql Example

Stopping this is trickier than it seems

Let's write some exciting database code to see why

Something for clowns

Sql Example

Stopping this is trickier than it seems

Let's write some exciting database code to see why

Something for clowns

```
input = grabInput()
request = ``select * from table1 where clown = ''
          + input + ``;``
info = db.run(request)
```

Sql Example

Now Sanitizing that is easy

Sql Example

Now Sanitizing that is easy

```
input = sanitize( grabInput() )  
request = ``select * from table1 where clown = ''  
          + input + ``;``  
info = db.run(request)
```

Sql Example

We know that is not how the code looks

Sql Example

We know that is not how the code looks
It is more like this:

Sql Example

We know that is not how the code looks
It is more like this:

```
input = grabInput()
unrelated_func()
munged_input = munge(input);
more_processing();
# a loop to distract you
for i from (1:n) {
    unrelated_func2(i)
}
sec_input = sanitize(munged_input);
request = ``select * from table1 where clown = ``
          + sec_input + ``;``
more_unrelated_work()
info = db.run(request)
```

Sql Example

It is also easy to get wrong

Sql Example

It is also easy to get wrong

```
input = grabInput()
unrelated_func()
munged_input = munge(input);
more_processing();
# a loop to distract you
for i from (1:n) {
  unrelated_func2(i)
}
input2 = sanitize(munged_input);
request = ``select * from table1 where clown = ''
          + input + ``;''
more_unrelated_work()
info = db.run(request)
```

Sql Example

- Can you find the error?

Sql Example

- Can you find the error?
- Could you find it at 4am?

Sql Example

- Can you find the error?
- Could you find it at 4am?
- And not paying full attention?

Sql Example

- Can you find the error?
- Could you find it at 4am?
- And not paying full attention?
- The program still runs

Sql Example

- Can you find the error?
- Could you find it at 4am?
- And not paying full attention?
- The program still runs
- Good enough right?

Sql Example

- Can you find the error?
- Could you find it at 4am?
- And not paying full attention?
- The program still runs
- Good enough right?

The Dynamic Approach

- We can run tests

The Dynamic Approach

- We can run tests
- See that our program can handle everything

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

But ...

- Needs deployable code

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

But ...

- Needs deployable code
- There will be code that isn't covered

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

But ...

- Needs deployable code
- There will be code that isn't covered
- Inputs will be missed

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

But ...

- Needs deployable code
- There will be code that isn't covered
- Inputs will be missed
- Requires diligence

The Dynamic Approach

- We can run tests
- See that our program can handle everything
- Check all the input
- You can do this in any language

But ...

- Needs deployable code
- There will be code that isn't covered
- Inputs will be missed
- Requires diligence
- Clutters the code

The Dynamic Approach

Wait? Extensive code coverage

Diligence and hardwork?

If we had that we wouldn't need those tests in the first place

The Solution

- Why not have a compiler do this?

The Solution

- Why not have a compiler do this?
- They yell at us when we pass ints to functions that expect strings

The Solution

- Why not have a compiler do this?
- They yell at us when we pass ints to functions that expect strings
- Can't they yell at us when we do something insecure

The Solution

- Why not have a compiler do this?
- They yell at us when we pass ints to functions that expect strings
- Can't they yell at us when we do something insecure

The Solution

They can yell at us!

The Solution

They can yell at us!

All we need is to encode security as a type

The Solution

They can yell at us!

All we need is to encode security as a type

Creating types is easy in most languages

Sql Example Revisited

This idea is at least twenty years old
Tom Moertel last year actually rediscovered this
So lets make a safe type:

```
class SafeString {
    private string str
    SafeString(String s) {
        str = sanitize(s)
    }
    toStr() {
        return str;
    }
}
```

Sql Example Revisited

Now strings that are secure have a type

Sql Example Revisited

Now strings that are secure have a type

So we can make functions only take data of that type

Sql Example Revisited

Now strings that are secure have a type

So we can make functions only take data of that type

```
String db.run(SafeString query);
```

Sql Example Revisited

So let's do that example securely

Sql Example Revisited

So let's do that example securely

```
input = grabInput()
unrelated_func()
munged_input = munge(input);
more_processing();
# a loop to distract you
for i from (1:n) {
  unrelated_func2(i)
}
request = ``select * from table1 where clown = ''
          + munged_input + ``;``
safe_request = SafeString(request)
more_unrelated_work()
info = db.run(safe_request)
```

Sql Example Revisited

- So what changed?

Sql Example Revisited

- So what changed?
- Not much, since this is working code

Sql Example Revisited

- So what changed?
- Not much, since this is working code
- It still works, as it should

Sql Example Revisited

- So what changed?
- Not much, since this is working code
- It still works, as it should
- Except for bad code, which will now fail

Sql Example Revisited

- So what changed?
- Not much, since this is working code
- It still works, as it should
- Except for bad code, which will now fail

Compiler Error: provided String expected SafeString

More Advantages

- Less code that must run to be tested

More Advantages

- Less code that must run to be tested
- All security checks done at compile-time

More Advantages

- Less code that must run to be tested
- All security checks done at compile-time
- Faster running code

More Advantages

- Less code that must run to be tested
- All security checks done at compile-time
- Faster running code
- No new fancy library or toolkit to learn

Extensions of Idea

This technique can be generalized

- This is obviously usable for XSS

Extensions of Idea

This technique can be generalized

- This is obviously usable for XSS
- Buffer Overflows can be protected with Generics (Shan & Kiselyov)

Extensions of Idea

This technique can be generalized

- This is obviously usable for XSS
- Buffer Overflows can be protected with Generics (Shan & Kiselyov)
- A modified form of this can be used to verify protocols (Adabi et al)

Extensions of Idea

This technique can be generalized

- This is obviously usable for XSS
- Buffer Overflows can be protected with Generics (Shan & Kiselyov)
- A modified form of this can be used to verify protocols (Adabi et al)
- Verify data structures and data formats

Limitations and Future Work

- Requires knowing you need to sanitize

Limitations and Future Work

- Requires knowing you need to sanitize
- Will not protect you from novel attacks

Limitations and Future Work

- Requires knowing you need to sanitize
- Will not protect you from novel attacks
- Needs to be done in a strongly typed compiling language

Limitations and Future Work

- Requires knowing you need to sanitize
- Will not protect you from novel attacks
- Needs to be done in a strongly typed compiling language
- Work on Capabilities and metalanguages can help

Questions?