Liquid Haskell
Haskell as a Theorem Prover

Niki Vazou
University of Maryland

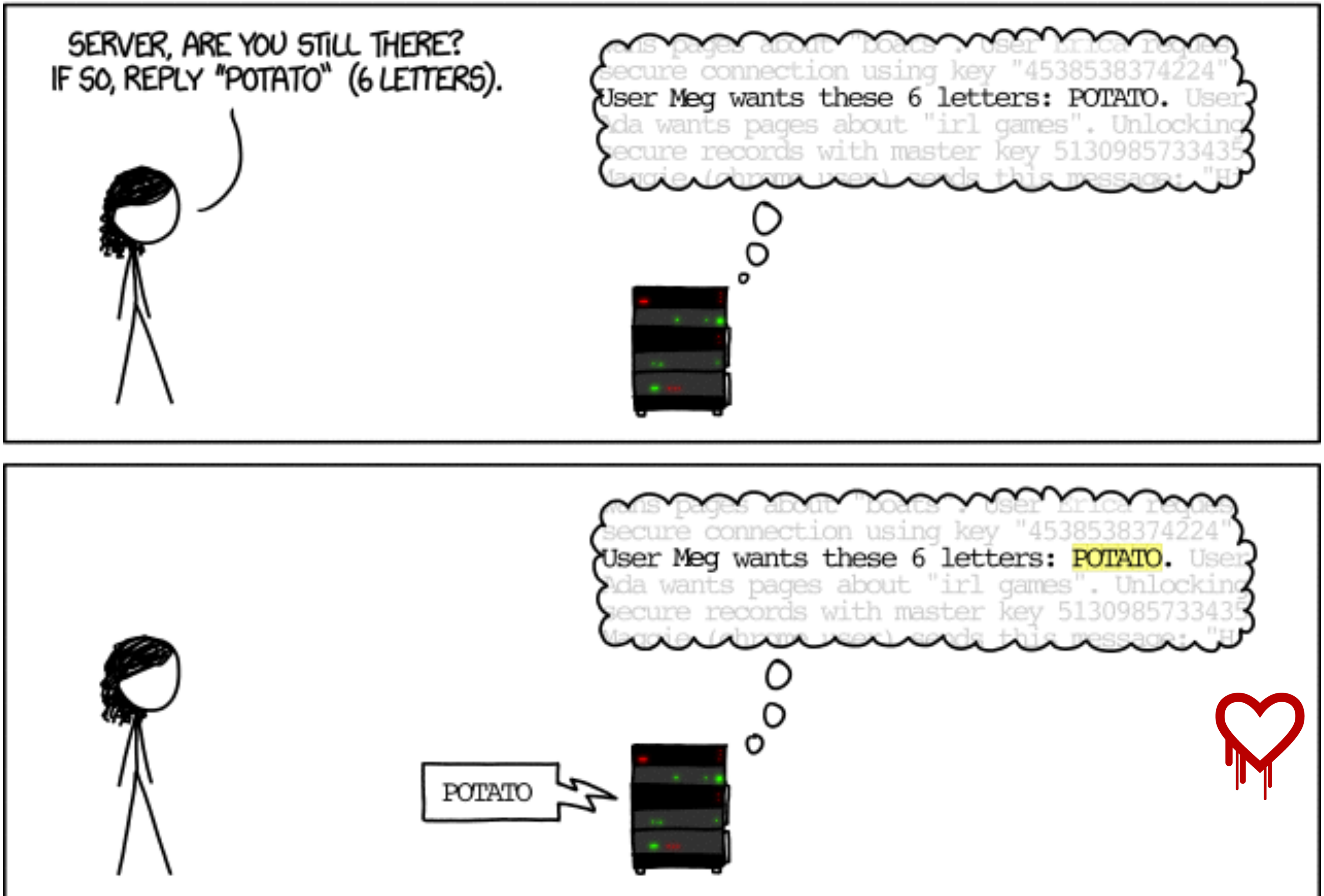
Software bugs are everywhere

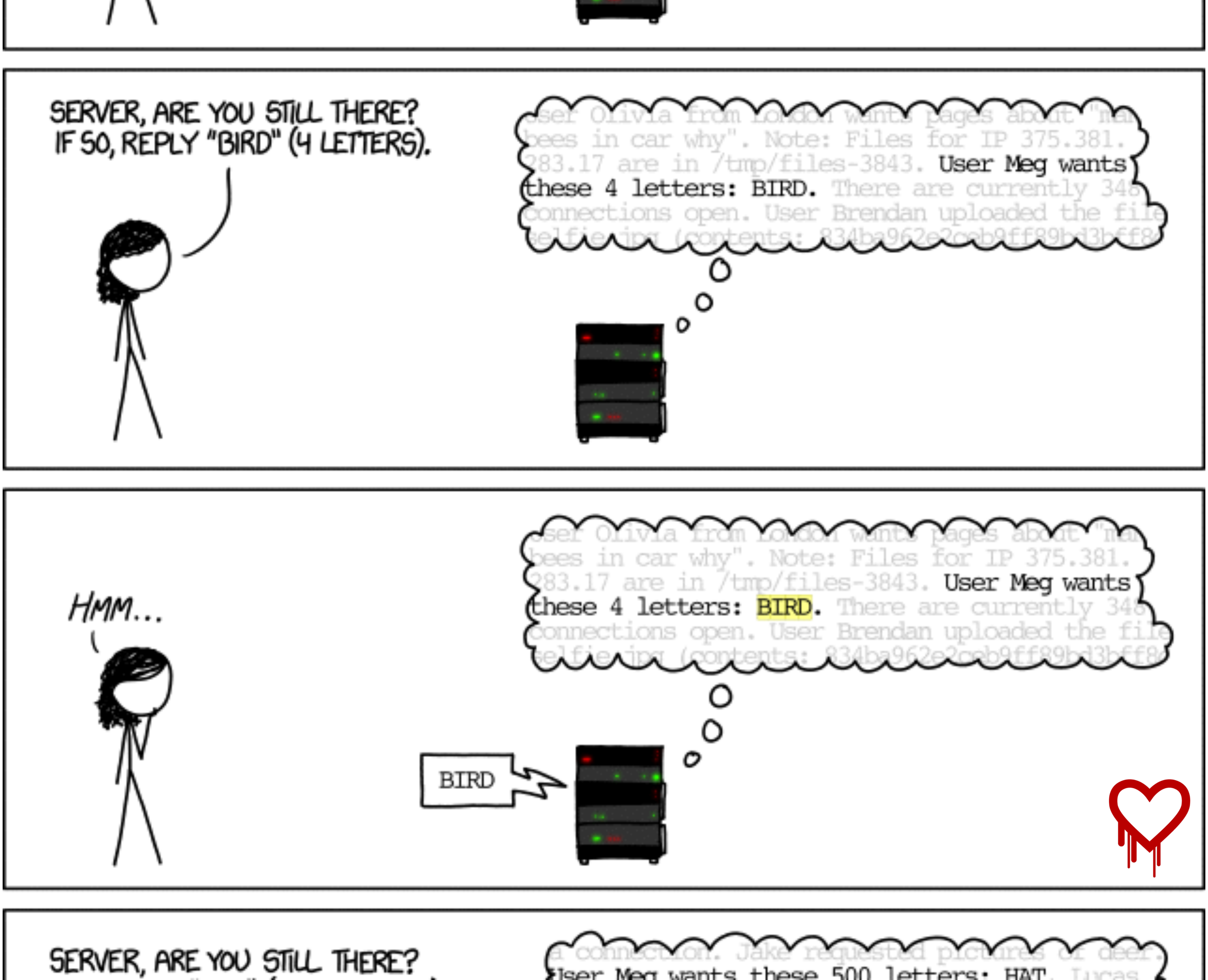
Airbus A400M crashed due to a software bug.
— May 2015

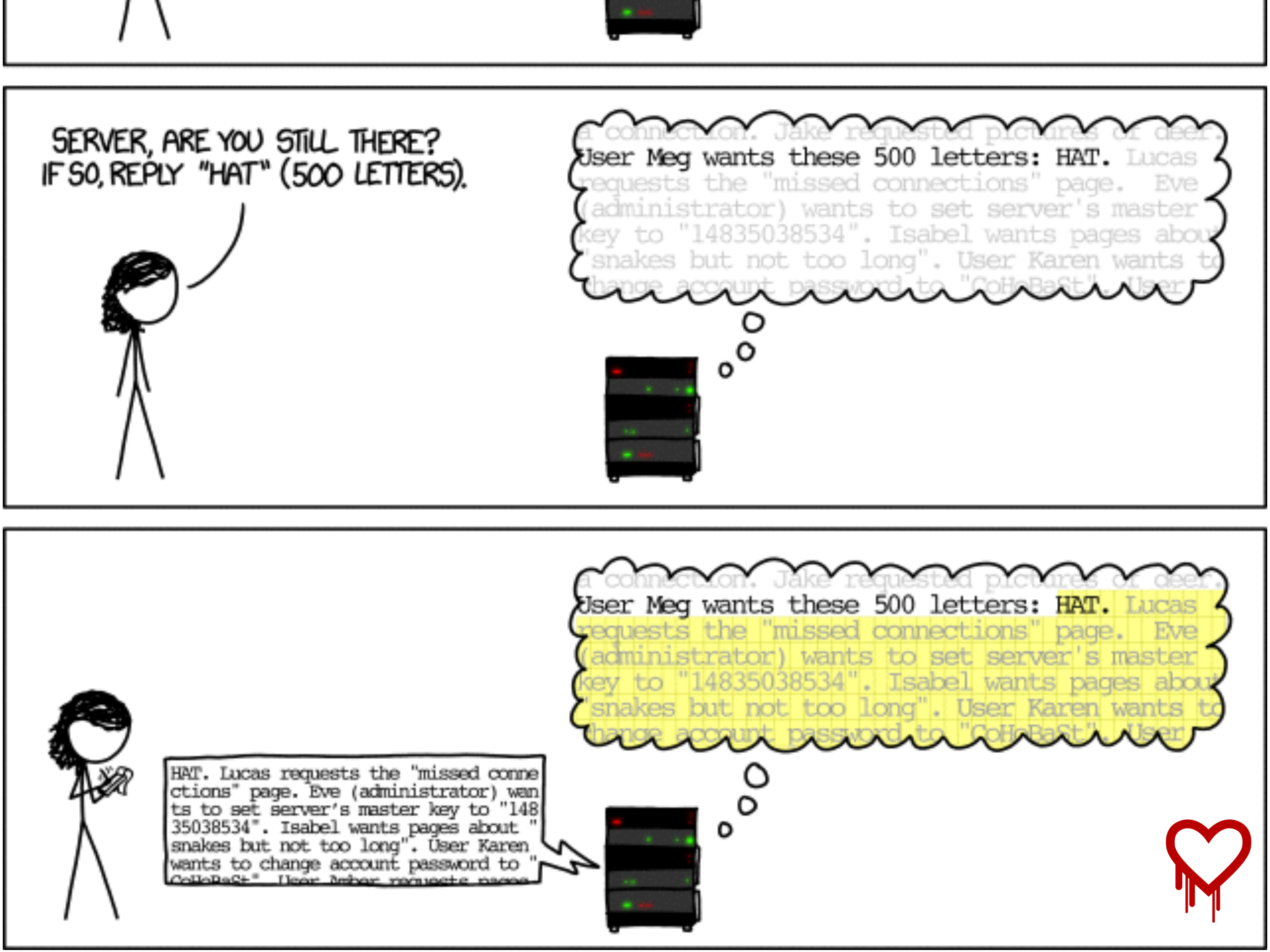
Software bugs are everywhere

The Heartbleed Bug.
Buffer overflow in OpenSSL. 2015

HOW THE HEARTBLEED BUG WORKS:




SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
ser Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
HMM...
ser Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD
SERVER, ARE YOU STILL THERE?
a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed conne ctions" page. Eve (administrator) wan ts to set server's master key to "148 35038534". Isabel wants pages about " snakes but not too long". User Karen wants to change account password to "

Make bugs difficult to express

Using Modern Programming Languages

F#, Ocaml, Erlang, Scala, Haskell

Because of

Strong Types + λ-Calculus

Make bugs difficult to express

Using Mode guages

F#, Ocar askell

Because of

ong Types + λ-Calculus

VS.

VS.

```
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> :t takeWord16
   takeWord16 :: Text -> Int -> Text
```

vs.

```
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> takeWord16 pack True
   Type Error: Cannot match Bool vs Int
```

VS.

```
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> takeWord16 pack 500
   “hat\58456\2594\SOH\NUL…
```


VS.

Valid Values for `takeWord16`?

```
takeWord16 :: t:Text -> i:Int -> Text
```

All Ints

…, -2, -1, 0, 1, 2, 3, …

Valid Values for `takeWord16`?

```
takeWord16 :: t:Text -> i:Int -> Text
```

Valid Ints	Invalid Ints
0, 1 … , len t	len t + 1, …

Refinement Types

```
take :: t:Text -> {v:Int | v <= len t} -> Text
```

Valid Ints	Invalid Ints
0, 1 … , len t	len t + 1, …

Refinement Types

```
take :: t:Text -> {v:Int | v <= len t} -> Text
```

```
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> take pack 500
   Refinement Type Error
```

Refinement Types

```
take :: t:Text -> {v:Int | v <= len t} -> Text
```

```
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> take pack 500
   Refinement Type Error
```

Liquid Haskell

Refinement Types

Checks valid arguments, under facts.

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text
heartbleed = let x = "hat"
             in  take x 500
```

```
len x=3 => v=500 => v<=len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x=3 => v=500 => v<=len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in take x 500
```

```
len x=3 => v=500 => v<=len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

$\texttt{len x=3} \Rightarrow \boxed{\texttt{v=500}} \Rightarrow \texttt{v<=len x}$

Checks valid arguments, under facts.

```
take :: t:Text -> {v | v <= len t} -> Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v <= len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x=3 => v=500 => v<=len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

SMT-query

```
len x=3 => v=500 => v<=len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

SMT-Invalid

$\boxed{\texttt{len x=3 => v=500 => v<=len x}}$

Checks valid arguments, under facts.

```
take :: t:Text->{v|v <= len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

Checker reports **Error**

Liquid Haskell:

Checks valid arguments, under facts.

What are interesting facts?

What are interesting facts?

```
len “hat” = 3
```

What are interesting facts?

```
len “hat” = 3

(xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

What are interesting facts?

```
len "hat" = 3

(xs ++ ys) ++ zs == xs ++ (ys ++ zs)

f is == mapReduce f is
```

What are interesting facts?

```
len “hat” = 3

(xs ++ ys) ++ zs == xs ++ (ys ++ zs)

f is == mapReduce f is
```

Theorems about Haskell functions

Theorems about Haskell functions

(Liquid) Haskell as a theorem prover.

(Liquid) Haskell as a theorem prover.

Specify theorems as Refinement Types

Prove theorems in Haskell

Use Liquid Haskell to check correctness

(Liquid) Haskell as a theorem prover.

```
data L a = N | C a (L a)
```

(Liquid) Haskell as a theorem prover.

```haskell
data L a = N | C a (L a)
```

```haskell
N ++ ys          = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys           = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
          ()
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys              = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
  {v:()|(xs ++ ys)++zs==xs++(ys++zs)}
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys            = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
```

(Liquid) Haskell as a theorem prover.

What is the body of assoc?

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ???
```

(Liquid) Haskell as a theorem prover.

What is the body of assoc?

A unit Haskell value showing that left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ???
```

(Liquid) Haskell as a theorem prover.

What is the body of assoc?

A unit Haskell value showing that left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ()
```

Type Error!

(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that
left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
  { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs
  =   (xs ++ ys) ++ zs
      …
  ==. xs ++ (ys ++ zs)
  *** QED
```

(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that

left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs
 =    (xs ++ ys) ++ zs
      ...
 ==. xs ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that
left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs
 =   (xs ++ ys) ++ zs
     …
 ==. xs ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that
left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs
 =   (xs ++ ys) ++ zs
     …
 ==. xs ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys          = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs
 =   (xs ++ ys) ++ zs


 ==. xs ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys            = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =   (N  ++ ys) ++ zs


 ==. N  ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys            = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =   (N  ++ ys) ++ zs


 ==. N  ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys              = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =    (N  ++ ys) ++ zs
 ==. ys          ++ zs
 ==. N  ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys             = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =   (N  ++ ys) ++ zs
 ==. ys ++ zs
 ==. N  ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys          = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =   (N  ++ ys) ++ zs
 ==. ys ++ zs
 ==. N  ++ (ys ++ zs)
 *** QED
```

(Liquid) Haskell as a theorem prover.

```
N ++ ys            = ys
(C x xs) ++ ys = C x (xs ++ ys)
```

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
 { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc N  ys zs
 =   (N  ++ ys) ++ zs
 ==. ys ++ zs
 ==. N  ++ (ys ++ zs)
 *** QED
```

Demo

(Liquid) Haskell as a theorem prover.

$\boxed{\textit{Demo}}$

(Liquid) Haskell as a theorem prover.

Specify theorems as Refinement Types

Prove theorems in Haskell

Use Liquid Haskell to check correctness

Thanks!

END