CS 208e
Reflections on Trusting Trust

Thursday, November 14th, 2019
Chris Gregg

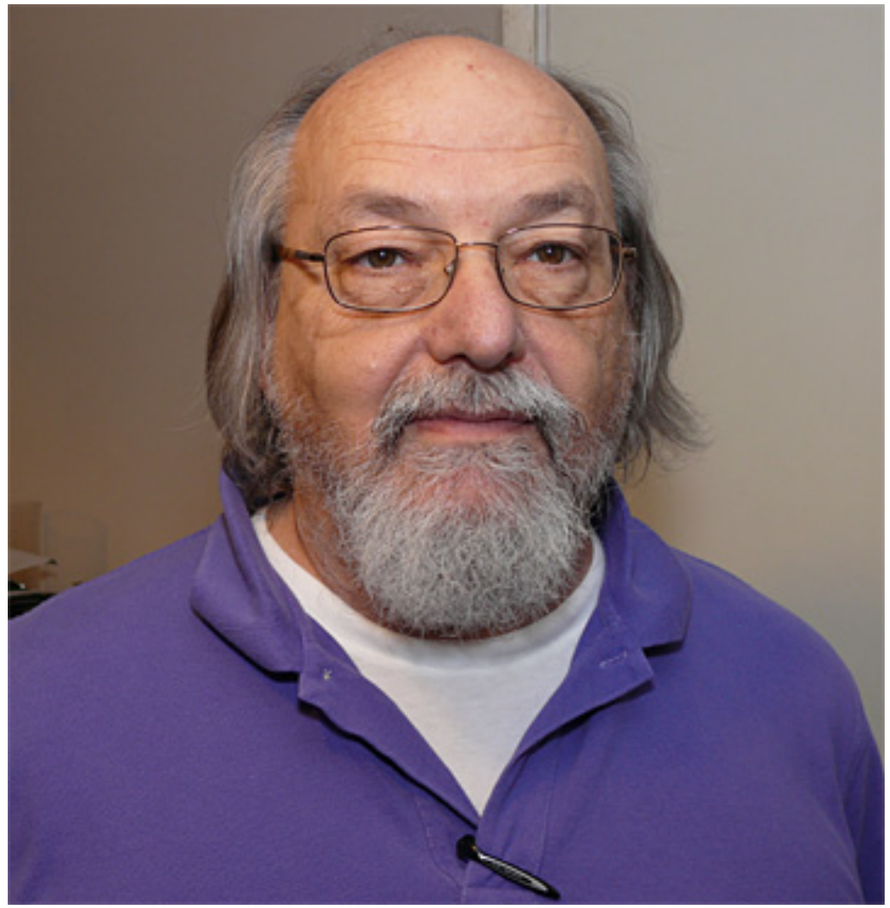
---

reading:
https://www.archive.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

Ken Thompson

- Ken Thompson is the creator of the Unix operating system, and a number of notable programming languages, including *B*, the predecessor to *C*, and of *Go* while at Google (where he still works).
- Most of his career was spent at Bell Labs, where he worked on Unix, and also made notable contributions to regular expression parsing, and the definition of the UTF-8 encoding scheme.
- He is a Turing award winner (1983, along with Dennis Ritchie), and this lecture will focus on his acceptance speech for that award, which is considered a seminal computer security paper.



Reflections on Trusting Trust - Stage I

- “I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.”

- “In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.”

- “More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source.”

- If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it.

Quines

- Thomson is talking about a *quine*, https://en.wikipedia.org/wiki/Quine_(computing)

From the Wikipedia article:

> A quine is a non-empty computer program which takes no input and produces a copy of its own source code as its only output. The standard terms for these programs in the computability theory and computer science literature are "self-replicating programs", "self-reproducing programs", and "self-copying programs".

- The name "quine" was coined by Douglas Hofstadter, in his popular science book *Gödel, Escher, Bach: An Eternal Golden Braid*

- Let's take a few minutes to try and write a quine! Use whatever language you want, and just jot down some code — your brain will stretch a bit from the exercise.

Chris's Mediocre Attempt

```
#!/usr/bin/env python

newline=chr(10)
quote=chr(39)
eq=chr(61)
a='#!/usr/bin/env python'
b='newline=chr(10)'
c='quote=chr(39)'
d='eq=chr(61)'
e='print(a+newline+newline+b+newline+c+newline+d+newline+chr(97)+e
q+quote+a+quote+newline+chr(98)+eq+quote+b+quote+newline+chr(99)+e
q+quote+c+quote+newline+chr(100)+eq+quote+d+quote+newline+chr(101)
+eq+quote+e+quote+newline+e)'
print(a+newline+newline+b+newline+c+newline+d+newline+chr(97)+eq+q
uote+a+quote+newline+chr(98)+eq+quote+b+quote+newline+chr(99)+eq+q
uote+c+quote+newline+chr(100)+eq+quote+d+quote+newline+chr(101)+eq
+quote+e+quote+newline+e)
```

A Very Concise Python Quine

```
s = 's = %r\nprint(s%%s)'
print(s%s)
```

```
$ python conciseQuine.py
s = 's = %r\nprint(s%%s)'
print(s%s)
```

You can check a quine in a unix system by piping the output back to another interpreter:

```
$ python conciseQuine.py | python | python | python
s = 's = %r\nprint(s%%s)'
print(s%s)
```

Ouroboros Quines and Multiquines

An *Ouroboros* quine, also known as a *quine-relay*, is a quine that is written in one language, outputs a program in *another* language, which, when run, outputs the original program in the original language.

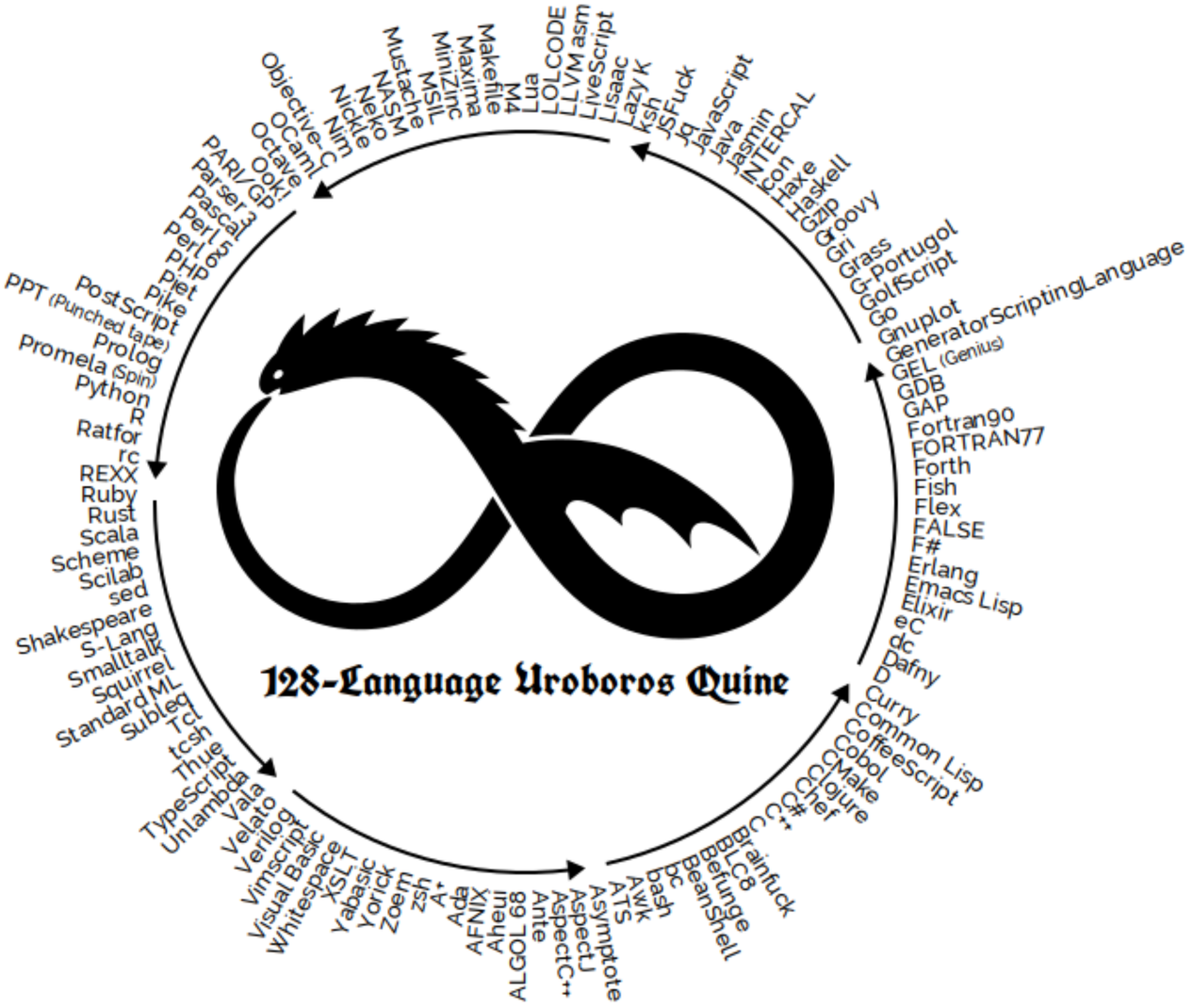
This can be extended to multiple levels of recursion. See the example from the lecture code.

From Wikjipedia:

> David Madore, creator of Unlambda, describes *multiquines* as follows:
> "A multiquine is a set of r different programs (in r different languages — without this condition we could take them all equal to a single quine), each of which is able to print any of the r programs (including itself) according to the command line argument it is passed. (Note that cheating is not allowed: the command line arguments must not be too long — passing the full text of a program is considered cheating)."

An Astounding Ouroboros Quine

https://github.com/mame/quine-relay



Reflections on Trusting Trust - Stage II

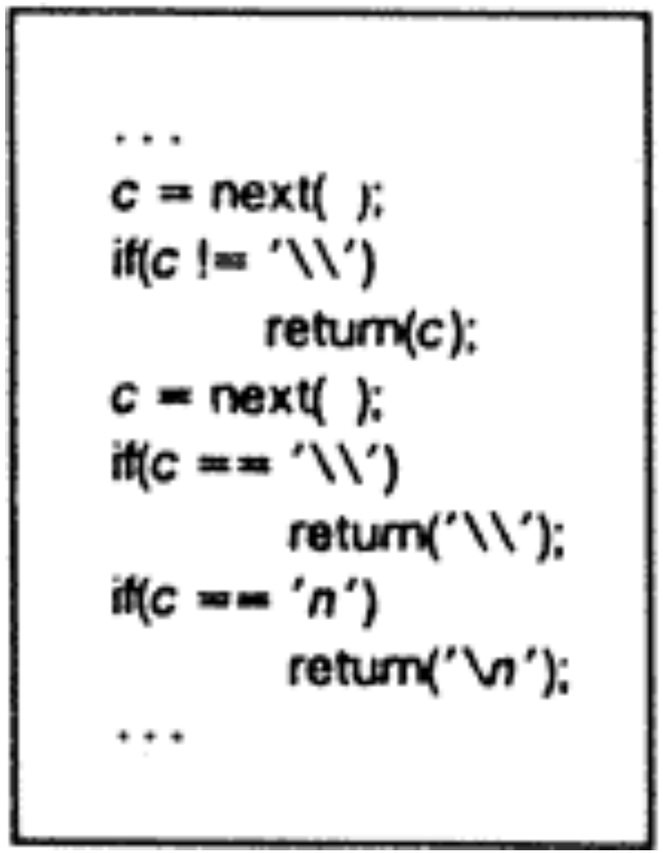
“The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this ease, I will use a specific example from the C compiler.

“C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

```
"Hello world\n"
```

represents a string with the character "\n," representing the new line character.”

Reflections on Trusting Trust - Stage II



```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

Figure 2

"Figure 2 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge."

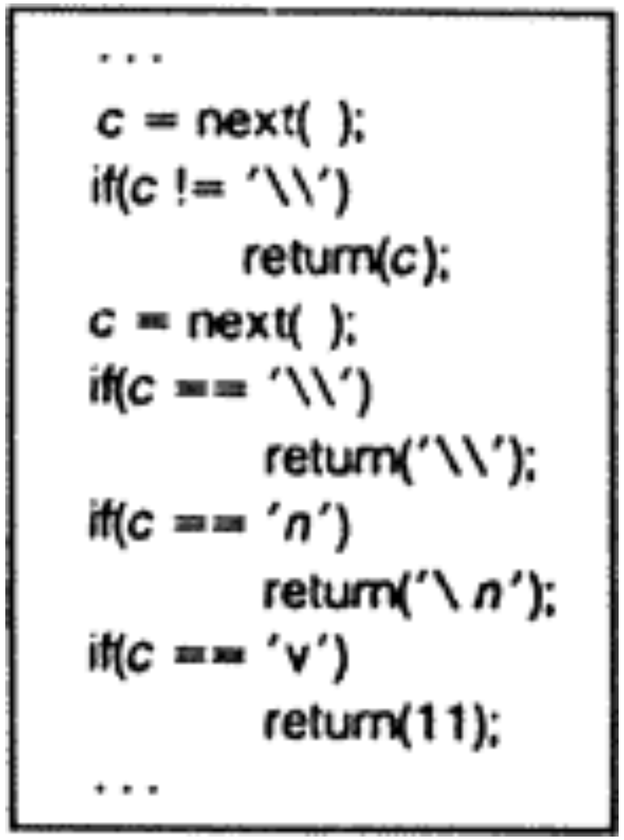
Reflections on Trusting Trust - Stage II

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

Figure 3

“Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2 is obvious and is presented in Figure 3. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 4. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 3.”

Reflections on Trusting Trust - Stage II

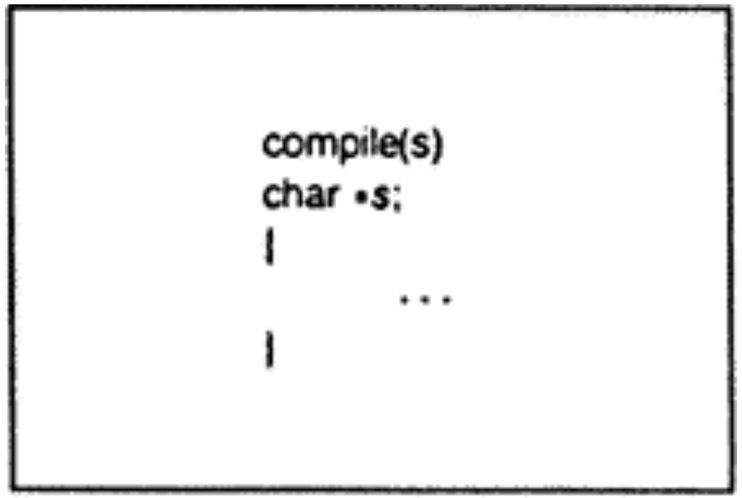


```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return(11);
...
```

Figure 4

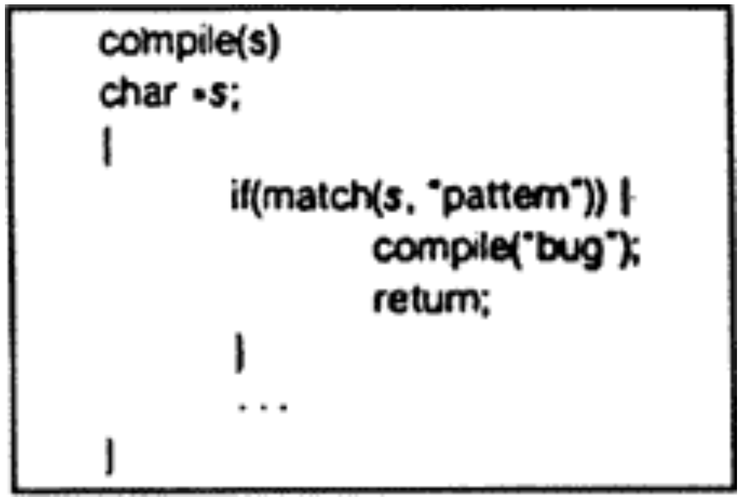
"This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition."

Reflections on Trusting Trust - Stage III



```
compile(s)
char *s;
{
        ...
}
```

Figure 5



```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

Figure 6

“Again, in the C compiler, Figure 5 represents the high-level control of the C compiler where the routine "compile" is called to compile the next line of source. Figure 6 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse." ”

Reflections on Trusting Trust - Stage III

> “The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.
>
> Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.”

This is a critical point: finding this Trojan Horse would be easy at this point — just look at the source code of the compiler, and there it is!

Reflections on Trusting Trust - Stage III

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

Figure 7

“The final step is represented in Figure 7. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere. ”

Reflections on Trusting Trust - Moral

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect.

Countering “Trusting Trust”

https://www.schneier.com/blog/archives/2006/01/countering_trus.html

“Wheeler explains how to defeat this more robust attack. Suppose we have two completely independent compilers: A and T. More specifically, we have source code SA of compiler A, and executable code EA and ET. We want to determine if the binary of compiler A -- EA -- contains this trusting trust attack.
Here's Wheeler's trick:
Step 1: Compile SA with EA, yielding new executable X.
Step 2: Compile SA with ET, yielding new executable Y.
Since X and Y were generated by two different compilers, they should have different binary code but be functionally equivalent. So far, so good. Now:
Step 3: Compile SA with X, yielding new executable V.
Step 4: Compile SA with Y, yielding new executable W.
Since X and Y are functionally equivalent, V and W should be bit-for-bit equivalent.
And that's how to detect the attack. If EA is infected with the robust form of the attack, then X and Y will be functionally different. And if X and Y are functionally different, then V and W will be bitwise different. So all you have to do is to run a binary compare between V and W; if they're different, then EA is infected.”

The Ken Thompson Hack in the Real World

https://nakedsecurity.sophos.com/2009/08/18/compileavirus/

References and Advanced Reading

- **References:**
  - https://www.archive.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf
  - http://wiki.c2.com/?TheKenThompsonHack
  - https://www.win.tue.nl/~aeb/linux/hh/thompson/trust.html
  - https://en.wikiquote.org/wiki/Ken_Thompson
  - https://en.wikipedia.org/wiki/Quine_(computing)
  - https://github.com/mame/quine-relay
  - https://www.schneier.com/blog/archives/2006/01/countering_trus.html

- **Advanced Reading:**
  - https://softwareengineering.stackexchange.com/questions/184874/is-ken-thompsons-compiler-hack-still-a-threat
  - http://www.madore.org/~david/computers/quine.html
  - https://nolancaudill.com/how-to-build-a-quine-eb717bfb7f1f
  - http://www.computerhistory.org/fellowawards/hall/ken-thompson/
  - https://www.youtube.com/watch?v=tc4ROCJYbm0
  - https://www.youtube.com/watch?v=JoVQTPbD6UY