CS615 - Aspects of System Administration

Writing System Tools

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/615/

Team Mission Link

Red Team: `https://is.gd/KnEKb9`

Blue Team: `https://is.gd/4Zqu94`

Black Team: `https://is.gd/7Z7RmV`

Green Team: `https://is.gd/vKpFqQ`, `https://is.gd/HCQ79v`

Typical SysAdmin Tasks

https://stevens.netmeister.org/615/scripting-exercise.html

Tools

Benefits of Automation

- *repeatability* – some of the tasks we perform are complex and need to be executed in a fixed order
- *reliability* – we are forgetful, make mistakes and tyops, think we know better, ...
- *regularity* – some tasks need to be executed at a specific time, some with a specific frequency, ...
- *flexibility* – minor variations of a complex task or the environment it executes in

Remember the Lego blocks: Breaking complex tasks into smaller components allows us to apply these benefits recursively.

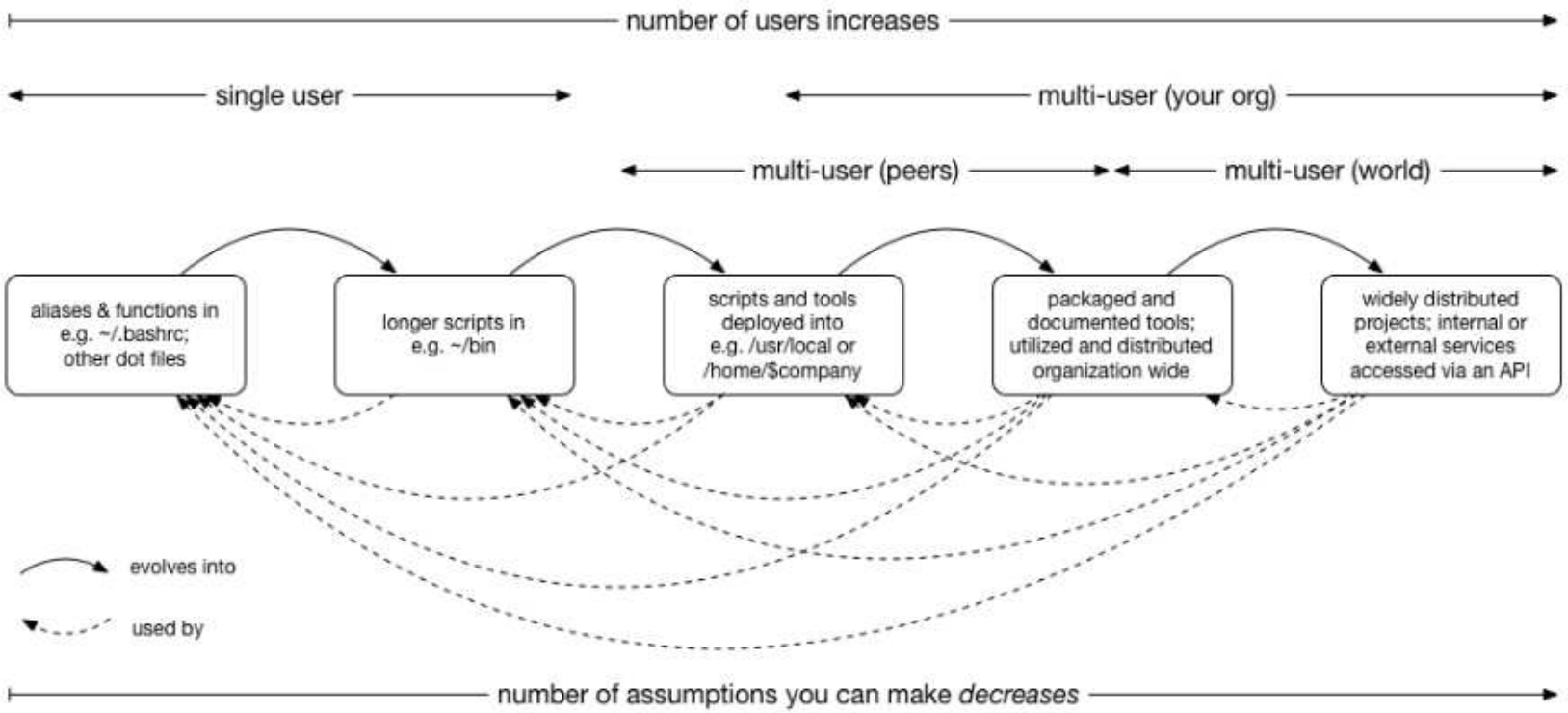
Words of Wisdom

Anything you do more than once
is worth automating.

Who benefits from automation?

- ourselves
- our peers
- our users
- our organization
- anybody anywhere

Evolution of SysAdmin Tools



Automation Pitfalls

- implementation and debugging takes time and effort
- automation often increases complexity
- automation increases the (potential for (negative)) impact
- automation may lead to a loss of audit trail or accountability
- abstracting a problem requires *understanding* it
- system tools require documentation

Note: some of these are hidden benefits!

Approaching Automation

Three basic categories:

- scripting
- programming
- software engineering

Approaching Automation

Three basic categories:

- scripting
  - automating *very* simple tasks
  - customization of user environment
  - often only suitable for one individual user
  - usually eventually evolves into larger programs

Approaching Automation

Three basic categories:

- scripting
  - automating *very* simple tasks
  - customization of user environment
  - often only suitable for one individual user
  - usually eventually evolves into larger programs
- programming
  - suitable for simple to moderately complex tasks
  - results frequently used by a small base of users
  - uses basic framework or common toolkits
  - provides consistent interface
  - may evolve into full product

Approaching Automation

Three basic categories:

- software development
  - required for any reasonably complex task
  - uses formal software engineering approach (measurable goals, requirements, specifications, ...)
  - may evolve from previous prototypes
  - requires ongoing continous maintenance / development efforts

The right tool?

Picking the right tool

Make sure to understand your requirements:

- motivation / goals
- target audience
- scope
- dependencies
- input / output requirements and contraints

The right tool?

Bourne shell (/bin/sh)

- lowest common denominator
- available and reliable on most platforms (but beware of non-portable bash(1) ”enhancements”)
- beware of ”quick-and-dirty” solutions, they grow to become unmaintainable
- treat shell as any other programming language:
  - use functions
  - use suitably scoped variables
  - follow Unix philosophy
  - properly package your tool

Note: you *can* “program” in shell, building complex systems.

The right tool?

Perl, Python, Ruby, Node, ...

- suitable for more complex tasks
- move to these when sed(1), awk(1), etc. become too cumbersome
- text manipulation and handling of well-defined data structures frequently easier
- beware of ”quick-and-dirty” solutions, they grow to become unmaintainable
- try to build self-contained modules that can be tested independent of the ”main” program
- wealth of libraries available – use them! (And remember to explicitly require them.)
- properly package your tool

Note: LOC is not a direct indicator of duct-tapeness.

The right tool?

Perl, PHP, Tcl, JavaScript, CoffeeScript, ...

- http/web server interfaces
- CGI ”scripts” / server-side execution
- interface with/utilize APIs in a specific domain/vendor products
- frequent cause of all sorts of security problems due to interface with user data / exposure on the internet

The right tool?

C, C++, Go, ...

- performance benefits
- portability
- sufficient low-levelness
- systems understanding
- fix/patch your other tools / the system

Interpreted Languages

General advantages:

- short development cycle
- normally facilitate things like string manipulation, arithmetic and more complex regular expressions
- easily handle multiple file handles and other I/O
- some security features
- tens of thousands of special- and general-purpose modules available

Interpreted Languages

General Disadvantages:

- no one tool fits all purposes
- tens of thousands of special- and general-purpose modules available
  $=>$ lots of duplication, stale code, questionable quality
  packaging and dependency resolution nightmares
- security features frequently neglected or circumvented (”too hard” or more precisely ”inconvenient”)
- everybody has their particular favorite (and dislikes one or the other)
- interpreter not (necessarily) universally available / installed

Regular Expressions

Example exercises:

- check if a string is a valid date
  `03/07/2016, 7.3.2016, 2016-03-07, 2016/03/07, 30/02/20, ...`
- extract all URLs from a document
  `http://example.com, ftp://example.com/dir/file.html,`
  `https://example.com/?foo=bar&blob=';alert(1);,`
  `http://example.com/?redir=http://example.com, ...`
- extract all proper words from a document
  `word, it's, Name, Henry the 3rd, cross-site scripting, ...`
- check if a string is a valid IPv4 address
  `0.0.0.0, 255.255.255.255, ...`
- check if a string is a valid IPv6 address
  `::1, fe80::e276:63ff:fe72:3900%xennet0, 2001:470:30:84:e276:63ff:fe72:3900, ...`

`https://stevens.netmeister.org/615/regex-exercise.html`

Regular Expressions

IPv4:

`https://regex101.com/r/Pgi7Rb/6`

Regular Expressions

IPv4:

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?
[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?\.(25[0-5]|2[0-4][0-9]|[01]
?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

Although this can be golfed down to:

```
^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}$
```

Regular Expressions

IPv4:

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?
[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?\.(25[0-5]|2[0-4][0-9]|[01]
?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

IPv6:

```
(([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,7}:|
([0-9a-fA-F]{1,4}:){1,6}:[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,5}(:
[0-9a-fA-F]{1,4}){1,2}|([0-9a-fA-F]{1,4}:){1,4}(:[0-9a-fA-F]{1,4}){1
,3}|([0-9a-fA-F]{1,4}:){1,3}(:[0-9a-fA-F]{1,4}){1,4}|([0-9a-fA-F]{1,
4}:){1,2}(:[0-9a-fA-F]{1,4}){1,5}|[0-9a-fA-F]{1,4}:((:[0-9a-fA-F]{1,
4}){1,6})|:((:[0-9a-fA-F]{1,4}){1,7}|:)|fe80:(:[0-9a-fA-F]{0,4}){0,4
}%[0-9a-zA-Z]{1,}|::(ffff(:0{1,4}){0,1}:){0,1}((25[0-5]|(2[0-4]|1{0,
1}[0-9]){0,1}[0-9])\.){3,3}(25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])|
([0-9a-fA-F]{1,4}:){1,4}:((25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])\.
){3,3}(25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9]))
```

Regular Expressions

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Better:

```
if (inet_pton(AF_INET, $ip)) {
        # AF_INET
} elsif (inet_pton(AF_INET6, $ip)) {
        # AF_INET6
} else {
        # not an IP address
}
```

Know when you need to be precise, and when 'good enough' is good enough.

Hooray!

5 Minute Break

Maybe play some Regex Golf?

`https://alf.nu/RegexGolf`

User Interface

https://www.netmeister.org/blog/consistent-tools.html

Unix Philosophy

Write programs that do one thing and do it well.

Write programs to work together.

Write programs to handle text streams, because that is a universal interface.

Be Boring

Use boring technology.

Write boring code.

Write *legible* code.

The Zen of Python

```
$ echo import this | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless
you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Readability counts!

Visual flow:

- use spaces/tabs/indentation *consistently*
- use a standard width terminal (~80 chars)
- refactor if code wraps / trails off right side
- refactor if logic doesn't fit into about one screen height
- avoid repeating the same code block

Readability counts!

Code is language:

- you are not charged per character
- use *descriptive* function and variable names
- use comments *where necessary*; explain *why*, not *what*
- don't use magic numbers
- write boring code

Pitfalls

- check the return value of any function that can fail!
- avoid file I/O whenever possible
- avoid using temporary files whenever possible
- don't assume you can write to the current working directory
- be explicit in setting permissions; set/use `umask(2)`
- use an exit handler to clean up after yourself
- retain idempotency whenever possible

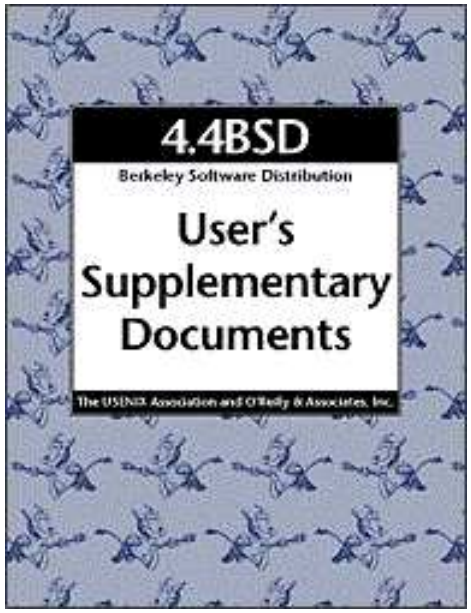
Never trust the user

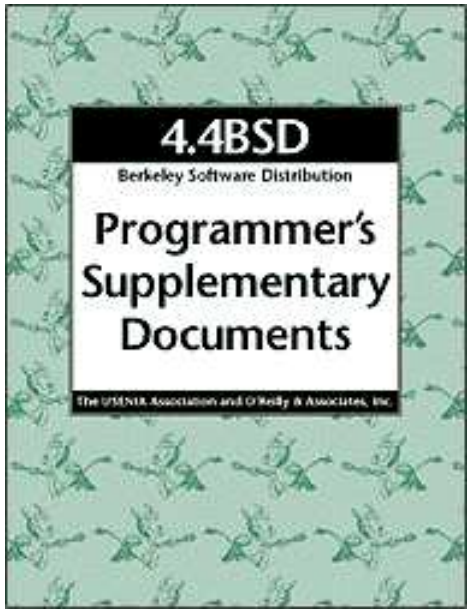
- your tool must be safe even if hostile input is given
- never trust the environment
- sanitize and validate all input
- you can't exhaustively identify all "bad" cases, so use a permit list, not a deny list

How to figure things out

- know your editor (tags, folds, jumping, ...)
- use a debugger
- use the source
- use `strace(1)`/`dtrace(1)`/`ktrace(1)` etc.
- write separate code to prove yourself right (or wrong)

Documentation





WTFM

Robustness Principle or Postel's Law

"Be conservative in what you do;
be liberal in what you accept from others."

Fail early, fail explicitly, fail gracefully.

POLA

Principle of Least Astonishment

Know your Users

Who will use your tools?

- you yourself only
- your peers
- your ”users”
- anybody else

What assumptions can you make?
How will your users script around your tool?
Can your tool be used as a filter?

Additional Accumulated Words of Wisdom

There's nothing as permanent as a temporary solution.

Don't let the Perfect be the enemy of the Good.

Release Early, Release Often.

"More users find more bugs."

F. Brooks, "The Mythical Man Month"

Learn to write a detailed bug report

Pre-requisite:

- RTFM
- Internet Search
- Know Your Community

Learn to write a detailed bug report

Pre-requisite: Do your homework.

Required:

- Description Of Problem
- Steps To Reproduce
- Expected Results
- Actual Results

Optional / recommended:

- Screenshots / *exact* copy of terminal I/O (`script(1)`)
- Suggested Remediation
- Code Patch

Know how to use your ticket tracking system!

- *value* problem reports
- link related tickets
- have a backlog strategy
- understand the lifetime of a ticket
  (reported, accepted, assigned, resolved, closed)
- abstain from passive-aggressive ticket closing and re-opening

Increase the Bus Factor

“Just friends.”

Collaboration is non-optional

Efficient use of Version Control Systems is a requirement. They allow you to:

- collaborate with others
- simultaneously work on a code base
- keep old versions of files
- keep a log of the who, when, what, and why of any changes
- perform release engineering by creating *branches*

Commit Messages

Commit messages are like comments: often useless and misleading, but critical in understanding human thinking behind the code.

Commit messages are full sentences in correct and properly formatted English.

Commit messages briefly summarize the *what*, but provide important historical context as to the *how* and, more importantly, *why*.

Commit messages SHOULD reference and integrate with ticket tracking systems.

See also:

- `https://is.gd/Wd1LhA`
- `https://is.gd/CUtwhA`
- `https://is.gd/rPQj5E`

Scope

"Contraints are friends."

F. Brooks, "The Mythical Man Month"

Scalability

- Simplify!
- Separate code and config.
- Reduce or eliminate interactions with the user.
- Premature optimization is the root of all evil.
- So is excusing shoddy programming.
- Fix *all* warnings and errors.
- Document all assumptions. Be specific.
- Always apply the Principle of Least Privilege.
- Assume hostile input and usage.
- Understand your code.
- Document your tools.

Program Maintenance

"... is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence."

F. Brooks, "The Mythical Man Month"

Toss it!

HW

https://stevens.netmeister.org/615/s20-ec2-backup.html

That's All, Folks!

Go away or I
will replace you
with a very small
shell script.

Reading

Shell:

- `http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
- `http://www.tldp.org/LDP/abs/html/`
- `http://sed.sourceforge.net/sed1line.txt`
- `csh(1)`, `ksh(1)`, `sh(1)`

Perl:

- `http://www.perl.com`
- `http://www.cpan.org`
- `perl(1)`, `perldoc(1)`, `perlfaq(1)`

Python:

- `http://www.python.org`
- pydoc

Reading

Ruby:

- http://www.ruby-lang.org/en/
- http://is.gd/cE7iFR
- http://is.gd/DR4aNU

Regex:

- https://alf.nu/RegexGolf
- https://regex101.com/
- https://regexr.com/
- https://www.regular-expressions.info/

Other:

- http://is.gd/jDDGpW
- https://www.netmeister.org/blog/writing-tools.html