HACKERMONTHLY Issue 18 November 2011

I Am Nothing

by Paul Buchheit

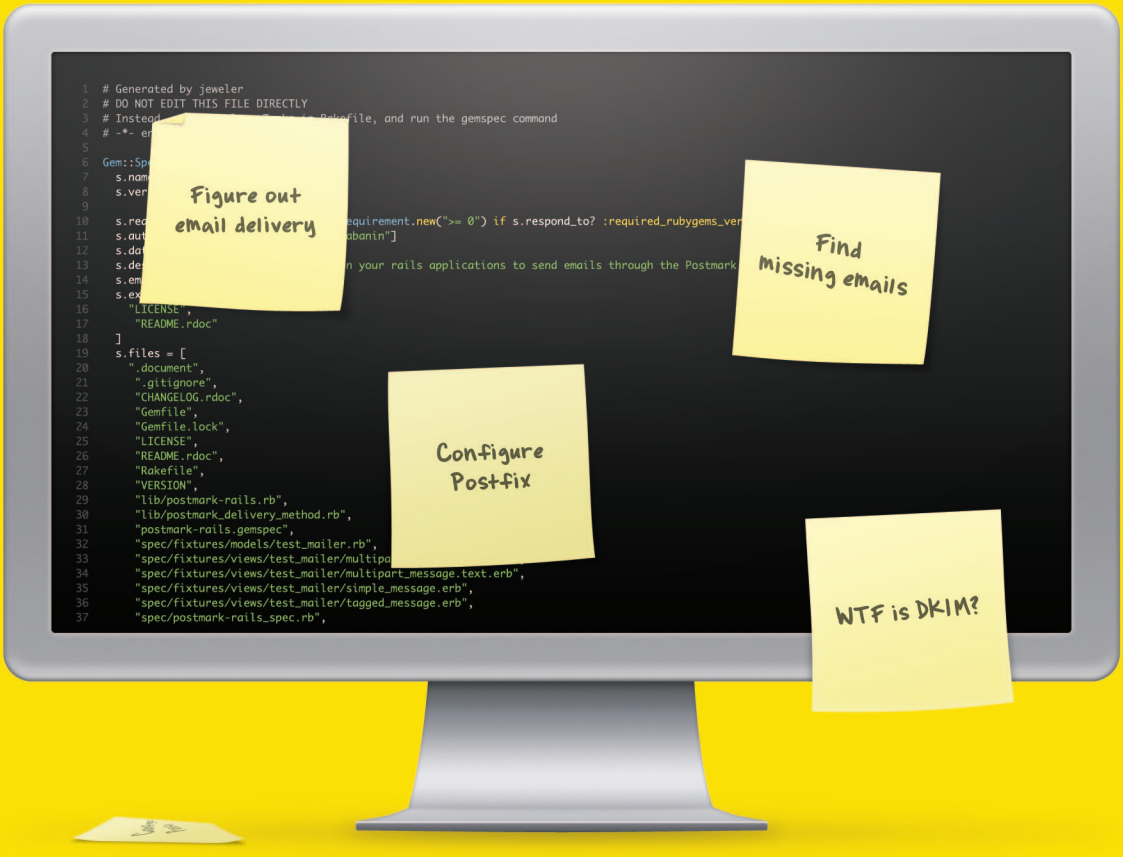
Our simple engagement tools help you understand your customers, prioritize feedback, and give great customer support even faster. Spend more time building a product your customers will love!

uservoice

Get 50% off your first 3 months* with the code **mindreader** at UserVoice.com.

* Offer good for new accounts if used before 12/31/2011.

If you enjoy configuring Postfix and Sendmail, ignore this.



We'll deal with that so you can get back to coding.

Postmark

hacker.postmarkapp.com

Transactional email delivery for web apps.

Curator
Lim Cheng Soon

Contributors
Paul Buchheit
Patrick McKenzie
Vinicius Vacanti
Chris Leary
Ferry Boender
Alan Skorkin
Michael Trick
Tony Haile

Illustrator
Matthew Phelan

Proofreader
Emily Griffin

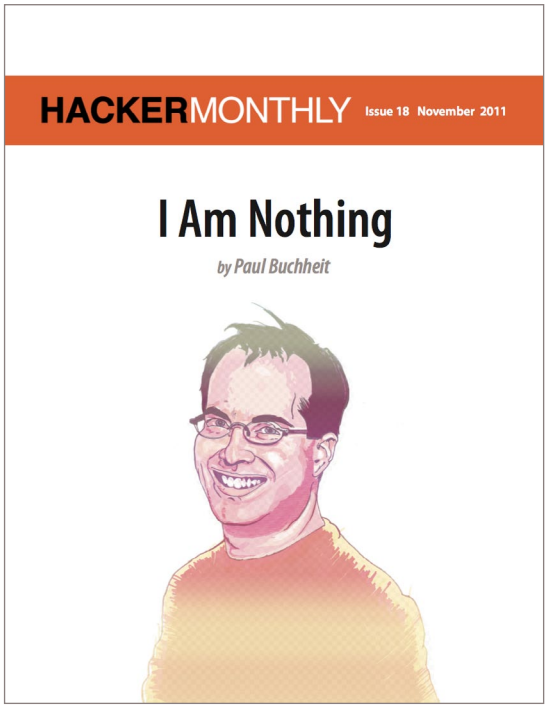
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*.

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Illustration: Matthew Phelan

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-18*

I Am Nothing

By PAUL BUCHHEIT

On a scale of one to ten, how good of a cog are you? How well do you function in your assigned role? How much of a man or woman are you? How do you rate yourself as a son or daughter, father or mother, wife or husband, heterosexual or homosexual, liberal or conservative, black or white, winner or loser, shark or sheep, introvert or extrovert, Christian, Muslim, atheist? How smart are you? How rational? How emotional? Do people like you? Are you getting ahead, or falling behind?

How do you know? Are you keeping an eye on the others in your category, comparing to see how you measure up to your peers? Is it more important for a man to be tall, or to have good hair?

This is, of course, the path of insanity, and not the good kind of insanity.

What will you do if you're too tough to be a good woman, too sensitive to be a good man, too selfish to be a good husband, too lazy to be a good employee, too shy to be a good friend, too caring to be rational, too fat to be pretty, too effeminate to be straight, too introverted to be a good leader, too smart to be kind, too young to be taken seriously, too old to make a difference, or too far behind to even get in the race?

These are all false standards and false dichotomies, but they are so common and so ingrained that we sometimes believe in them without even realizing it. And this leads to a mountain of insecurities, because nobody measures up to these crazy standards (and nobody should). But even if we don't believe in these things, it still matters what other people think, right? What will the neighbors think? Or how about our co-workers, or the people at church? And so everyone works to hide their insecurities, and they look around at their peers for comparison, and maybe they feel bad because everyone else seems to have it easy, to have it all figured out. The truth is nobody can see the truth anymore. They are all working to hide the truth: that they are afraid of who or what they really are. So they all put on a show, and they pretend to be a good whatever. Or maybe they rebel, and make a point of being a bad whatever, but then they are still under the control of that false standard, and they are still not being themselves.

That is all so exhausting.

I am nothing. It's simple. If I were smart, I might be afraid of looking stupid. If I were successful, I might be afraid of failure. If I were a man, I might be afraid of being weak. If I were a Christian, I might be afraid of losing faith. If I were an atheist, I might be afraid of believing. If I were rational, I might be afraid of my emotions. If I were introverted, I might be afraid of meeting new people. If I were respectable, I might be afraid of looking foolish. If I were an expert, I might be afraid of being wrong.

But I am nothing, and so I am finally free to be myself.

This isn't license to stagnate. Change is inevitable. Change is part of who we are, but if we aren't changing for the better, then we are just slowly decaying.

By returning to zero expectations, by accepting that I am nothing, it is easier to see the truth. Fear, jealousy, insecurity, unfairness, embarrassment — these feelings cloud our ability to see what is. The truth is often threatening, and once our defenses are up, it's difficult to be completely honest with anyone, even ourselves. But when I am nothing, when I have no image or identity or ego to protect, I can begin to see and accept things as they really are. That is the beginning of positive change, because we cannot change what we do not accept nor understand. But with understanding, we can finally see the difference between fixing problems and hiding them, between genuine improvement and faking it. We discover that **many of our weaknesses are actually strengths once we learn how to use them, and that our greatest gifts are often buried beneath our greatest insecurities.**

> **"By returning to zero expectations, by accepting that I am nothing, it is easier to see the truth."**

Letting go of your identity can be difficult and takes time, possibly forever. But as with any change, **moving in the right direction is all that really matters** (which is why you shouldn't compare yourself with others; you didn't start in the same place or with the same challenges). Fortunately, we have a variety of emotions that can help us: pride, anger, fear, jealousy, insecurity, unfairness, embarrassment, bitterness, etc. These are sometimes portrayed as bad emotions, but there's no such thing as a bad emotion, just bad responses to emotions. (For example, torturing children is a very harmful response to fears about your own sexuality) Instead, **use these emotions as a cue to remember that "I am nothing."** When you let go of your identity and examine why you are feeling the emotion (typically because something has threatened your identity), then these emotions are beneficial. They reveal the truth.

True self improvement requires becoming a better version of our selves, not a lesser version of someone else. But without self acceptance and understanding, how can we even know what that looks like or whether we're headed in the right direction? It would be like putting the final touches on the Mona Lisa while picturing some celebrity you saw on the cover of People magazine: the result would be a mess. Until we let go of our mental images of who we are or who we should be, our vision remains clouded by expectation. But when we let go of everything, open ourselves to any truth, and see the world without fear or judgment, then we are finally able to begin the process of peeling off the false identity that prevents our true self from growing. And it starts with nothing. ■

Paul Buchheit is a partner at the venture capital firm Y Combinator. He previously co-founded FriendFeed, which was acquired by Facebook in 2009, and was one of the first engineers at Google. At Google, he started Gmail, suggested the "Don't be evil" motto, and created the first AdSense prototype. Paul has a degree in Computer Science from Case Western Reserve University.

Reprinted with permission of the original author. First appeared in *hn.my/nothing* (paulbuchheit.blogspot.com)

Software Businesses in 5 Hours A Week

By PATRICK MCKENZIE

❶ Charge More Money

Most engineers severely undercharge for their products. This is particularly true for products which are aimed at businesses — almost all SaaS firms find that they make huge portions of their revenue from the topmost plan which is bought by people spending other people's money, but instead of optimizing for this, we opt for charging "fair" prices as determined by other software developers who won't pay for the service anyway. This is borked. Charge more.

❷ Do Web Applications

Faster iteration is a big deal. The faster you can deliver product to your customers, the faster you can get changes to your customers, the faster you learn about your business. And the better your software will get, the happier your customers will be, the more money you will make.

You get higher conversion rates to web applications in many cases. Mine is double what the downloadable application used to be. There are many, many things that could go wrong with downloading an application. You download it, "Where'd the installer go? I don't know how to install things. If I install this on my computer, will it steal my documents and break my Googles?" Common customer worries. If it's just a website, they won't have that worry.

Web applications that build recurring revenue are always a great thing. You have funnels leading up to your web application, and you also have funnels within your web application. One of the things that I track religiously is someone signing up for a free trial of my Bingo Card Creator. Do they actually get Bingo cards spitting out of their printer? If they don't, I have failed in some way. Maybe my software is too complicated to use, and I could talk for an hour about this, about little optimizations I've made to the internals of my application to make it more likely that they succeed in getting their job done for tomorrow. And as you make it more likely that people are going to succeed with using your software, you'll see the number of people who convert and the time that they stay using the software will increase, and that's money Straight To Your Bottom Line.

❸ Put More of Your Iceberg Above the Water Line

Businesses create value with almost everything they do. The lion's share of the value is, like an iceberg, below the waterline: it cannot be seen from anywhere outside the business. Create more value than you capture, certainly, but get the credit for doing so, both from Google and from everybody else. This means producing value in a public manner. Did you write software outside the core of your line of business? Great, OSS it. Get the credit. Have you learned something novel about your customers or industry? Write about it. Get the credit. Are your business' processes worthy of emulation? Spread what you know. Get the credit.

❹ SEO

I mentioned obliquely that half of my sales come from SEO, and 75% of my profits do. As developers who are trying to get into marketing, this is the thing that you will learn most easily and will make you huge, huge amounts of money if you do it well.

The biggest SEO problem that entrepreneurs have is this: you have a website consisting of five or six pages, and there's no reason for someone to cite that website unless they are in a commercial relationship with you. So if they use your software, they've paid their money, and they're happy, maybe they'll blog about that, and that's good. Getting money from someone is very hard. Getting them to cite you is less hard if you can produce something of value for them.

❺ Optimize Everything

Some of the most important advice I ever heard regarding the software business came from Steve Pavlina: all factors in the success of a software business are multiplicative. So if your conversion to the trial goes up by 10%, and your conversion to the sale goes up by 10%, you don't go up by 20% to your bottom line, you go up by 21%, because 1.1 x 1.1 is 1.21. So if you just get a 5% increase every month for a year, you get 70% growth in revenue. Yeah, it's a hill-climbing algorithm. Yeah, it takes some time, and it's not going to give you the 10x, 100x, 1000x return that some people are looking for, but I hill-climbed all the way out of the day job from hell, so it's an option.

First track how many of your users never come back, and you will find it is a scary, scary number. I've been optimizing this for years. My number is 60%. So I'm paying Google thousands of dollars a month, and 60% of thousands is totally wasted because they never come back after the first time. Lower that number by making their first experience, their first five minutes with the software totally awesome. Getting them to that point with activation will produce great returns.

❻ Outsource/Automate/Eliminate So You Can Do It All in 5 Hours a Week

Three ways to avoid wasting your time: outsource, automate, and eliminate.

- **Outsourcing.** Outsourcing means that you delegate tasks to be done by other people without harming the value too much.
- **Automation.** Have the computer do it, especially for repetitive things.
- **Elimination.** If it doesn't add value, then you shouldn't be doing it.

What to Outsource:

- **Web design.** I have a seven to eight day schedule for getting my website up. I'm not going to both make an application and hack together a website in that time. Web development talent is really cheap right now, like web design talent. So hand off the work to people who are talented, who like doing this stuff, and who constantly under price themselves. Let them make the websites. You do the work that adds value uniquely to your business that you can't get done by other people.
- **Web content.** How many people have written every word of text on their websites? There are many copywriters who can do that for you, so hire them to do it, because they're cheaper than you are. The end goal is maximizing effectiency. And any time you are performing a task that can be outsourced for far lower than your goal wage without compromising quality and without compromising your users' trust in you, then it should be done by someone else.
- **Self-contained programming projects.** If it can be completed by someone else and checked by you in an efficient fashion, then delegate.

Don't do all the development by yourself just because you can. Your success will not be determined by the number of lines of code you write. Write code when you want to write code, when it makes you happy, because you should be happy when you're running your own business. Do not work at any soul-sucking job for 19 hours a day. Key takeaway: if a Japanese company ever offers you a salaried position, just say "No!"

What to Automate:

- **Routine customer support tasks.** You will find that the same four things are taking all of your time. In fact, when I was selling downloadable software, I dealt with the following: "What is my registration key? I forgot it." "The Googles ate my computer." "A virus ate my hard drive, it's no longer installed. Can I get back? Where is my registration?"

- **If something comes up more than three times, automate that.** Any support tasks that I've ever done three times can be done with like one click from my dashboard. For example, I didn't used to issue receipts. Who needs a receipt, right? Well, people who want to get reimbursed need receipts. So the first two times I hand-wrote a receipt for her in notepad: ### This is a receipt, not an invoice ###. Here's your name, here's my name. You paid me $29.95. This is your receipt. And I e-mailed that. I did it like three times. And the fourth time, I'm like, "Should the CEO really be writing receipts by hand in Notepad?" No. So I wrote software to do that, and there's one button that I can click that will send you a receipt.
- **Drudgery.** License generation. If that isn't outsourced or automated already, it should be.
- **Server maintenance and monitoring.** I used to check my server every morning when I got up just to make sure it had not died in the middle of the night. That's very repetitive. It's important that I do it because if it dies, then, oh dear! But computers can do that much easier than I can, and they can check every five minutes of the day and just sent me an alert Iif my server goes down.

What to Eliminate:

- **Checking Google Analytics 37 times a day.** I used to do it. I wasted a lot of time on that. I had five hours a week and I'd spend one hour on Google Analytics learning nothing. Be honest with yourself. Is what I'm doing right now really driving business forward? If not, don't do it.
- **Worrying about competition.** There are fifteen other people who have done bingo card creation software. Many of them have cloned me, soup to nuts. So I guess their marketing strategy is being Patrick McKenzie-like. That doesn't work out so well. So don't worry about your competition. They'll clone you or they won't clone you. Who cares? Do right by your customers. It will work out in the end.
- **Development which is not meeting customer needs.** I spent twenty hours making this one feature for my software — that's an entire month of my business sucked up. I thought they had this problem: my teachers clearly don't understand where a file is. I've been talking to them for a couple of years, they don't get what the file system metaphor actually means. They want their bingo cards they make at home to be available at school, so I made a way that they could upload their bingo cards to my website and then download them from school. I never asked an actual teacher, "Would you use this?" ■

Patrick McKenzie runs a small software business. His current focus is on Appointment Reminder, which solves small businesses' problems with missed appointments. He also made Bingo Card Creator and consults from time to time, mostly on software marketing.

Reprinted with permission of the original author. First appeared in *hn.my/5hours* (kalzumeus.com)

Summary

1. **Charge More Money**
2. **Do Web Applications**
3. **Put More of Your Iceberg Above the Water Line**
4. **SEO**
5. **Optimize Everything**
6. **Outsource**
   - Web design
   - Web content
   - Self-contained programming projects

Automate

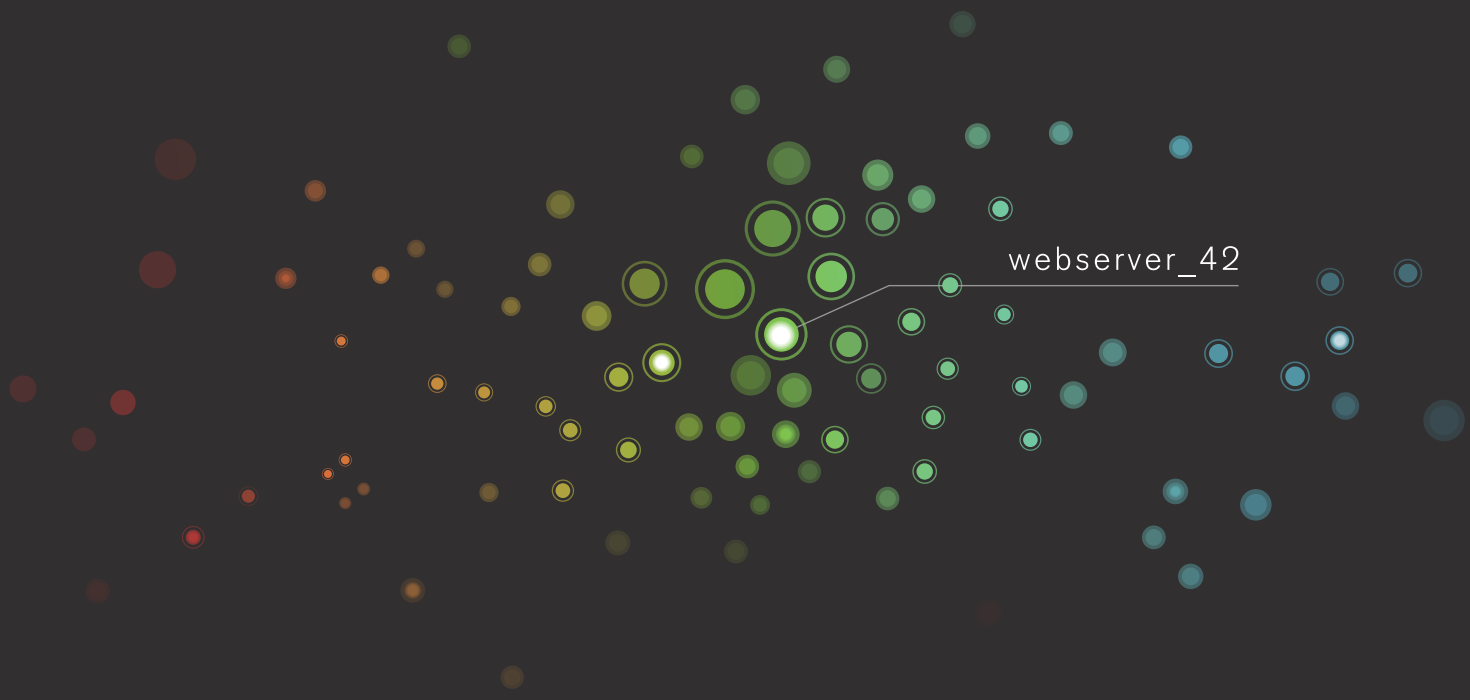
- Routine customer support tasks
- Something that comes up more than three times
- Drudgery
- Server maintenance and monitoring

Eliminate

- Checking Google Analytics 37 times a day
- Worrying about competition
- Development which is not meeting customer needs

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

The Long Grind Before You Become an Overnight Success

By VINICIUS VACANTI

"So, what do you do?"

Ugh. I hated that question.

The truth was that we were trying to start a new venture, but we hadn't really made any progress.

But, instead of just muttering something, I would force myself to enthusiastically pitch our current struggling idea. They would nod along, but the skepticism on their face was hard to ignore.

And, when I was done, they would sometimes hit me with: "So, is that your full-time thing?" Ugh. What that really meant was: you're trying to tell me that you spend all your time working on that ridiculous idea?

The Grind

We left our finance jobs in the summer of 2007, and we worked really, really hard. By February of 2010, it had been over two and half years of hustling on no salary. What did we have to show for it? Nothing.

We hadn't made a dollar of revenue. We had been rejected by every investor we talked to. We hadn't been able to recruit anyone to join our team. We hadn't gotten traction with any of our ideas.

We had failed to get more than 10K monthly unique visitors for Yipit for the last two years despite trying several ideas with it. We were going sideways.

On a personal level, my life savings was disappearing. I kept getting hit with late penalties on my credit card. Not because I didn't have the cash to pay it, but because I just didn't want to think about it. It was too depressing to look at my depleting bank account that I had worked so hard to build up. I remember withdrawing all the money from my 401K account and having to confirm that I did, in fact, understand the massive penalties I would incur for doing so.

In all honesty, I probably would have given up earlier. The only reason why I didn't was out of loyalty to my co-founder, Jim, who had also quit his finance job. He had passed up many amazing job opportunities to work alongside me, and I wasn't going to quit on him.

Everything Changes

In February of 2010, over two and half years since we started, we have yet another idea: build an aggregator for the early but quickly growing daily deal industry. The idea was sound, timely, and right up our alley, since we had been doing local deal aggregation for the last nine months.

And, in just three days, everything changed.

We launched the new idea in a three-day scramble, got some initial press, users loved it, and four months later raised $1 million from amazing investors. A year after that, we've raised $6 million, made real revenue, attracted hundreds of thousands of users, and recruited amazing people to join our team. And, best of all, we're just getting started.

> **While we didn't have outward signs of success, we had learned something very important: the art and science of starting a new venture.**

So, what happened in those three days?

I'm convinced that if we had the idea for a daily deal aggregator back in 2007 or 2008 or even 2009, we wouldn't have gotten traction because we would have messed it up.

But, after two and half years of failing and learning, we knew exactly what to do:

- **Product strategy.** We had become a part of the lean startup movement. I had gone to the New York lean startup meetups from the beginning, read Four Steps to Epiphany, and knew we just needed to build a minimum viable product.
- **Coding the prototype.** I had taught myself web development over the last few years, and Jim had taught himself front-end development. We didn't need to find an outsourcer, we just quickly built it ourselves.
- **Designing the user interface.** We had already designed a bunch of prototypes. We knew how to design a landing page that collects user email addresses and a sign-up flow that collects preferences, and we knew to ask our new users to spread the message.
- **Getting initial press.** We knew how to craft our story in a way that would get journalists interested. We got featured on TechCrunch and Wired, giving us a strong initial boost.
- **Getting investors interested.** We had built relationships with many New York angel investors over the last few years, and so we were able to quickly drum up some interest based on our traction since they already knew who we were.
- **Building buzz.** We had become involved in the New York tech community and our friends in the industry really helped us build initial buzz for Yipit.

Now that I look back, I realize that I was wrong to think that we had nothing to show for two and half years of hustling. While we didn't have outward signs of success, we had learned something very important: the art and science of starting a new venture. It took us almost three years to know what exactly we had to do during those three days.

And so, to everyone out there who's struggling and feels like they have nothing to show for it, I hope this article keeps you going. You're learning every day. And, when the inspiration strikes, you're going to be ready to pounce on it. ■

Vinicius Vacanti is the co-founder and CEO of Yipit, which aggregates and recommends daily deals based on your category and location preferences. Previous to startups, Vin worked as an investment analyst on Wall Street. He graduated from Harvard College with a degree in Applied Mathematics.

Reprinted with permission of the original author.
First appeared in *hn.my/grind* (viniciusvacanti.com)

You've Got To Find What You Love

Stanford Commencement Address, June 2005
By Steve Jobs

I AM HONORED TO be with you today at your commencement from one of the finest universities in the world. I never graduated from college. Truth be told, this is the closest I've ever gotten to a college graduation. Today I want to tell you three stories from my life. That's it. No big deal. Just three stories.

The first story is about connecting the dots.

I dropped out of Reed College after the first 6 months, but then stayed around as a drop-in for another 18 months or so before I really quit. So why did I drop out?

It started before I was born. My biological mother was a young, unwed college graduate student, and she decided to put me up for adoption. She felt very strongly that I should be adopted by college graduates, so everything was all set for me to be adopted at birth by a lawyer and his wife. Except that when I popped out they decided at the last minute that they really wanted a girl. So my parents, who were on a waiting list, got a call in the middle of the night asking: "We have an unexpected baby boy; do you want him?" They said: "Of course." My biological mother later found out that my mother had never graduated from college and that my father had never graduated from high school. She refused to sign the final adoption papers. She only relented a few months later when my parents promised that I would someday go to college.

And 17 years later I did go to college. But I naively chose a college that was almost as expensive as Stanford, and all of my working-class parents' savings were being spent on my college tuition. After six months, I couldn't see the value in it. I had no idea what I wanted to do with my life and no idea how college was going to help me figure it out. And here I was spending all of the money my parents had saved their entire life. So I decided to drop out and trust that it would all work out OK. It was pretty scary at the time, but looking back it was one of the best decisions I ever made. The minute I dropped out I could stop taking the required classes that didn't interest me, and begin dropping in on the ones that looked interesting.

It wasn't all romantic. I didn't have a dorm room, so I slept on the floor in friends' rooms, I returned coke bottles for the 5¢ deposits to buy food with, and I would walk the 7 miles across town every Sunday night to get one good meal a week at the Hare Krishna temple. I loved it. And much of what I stumbled into by following my curiosity and intuition turned out to be priceless later on. Let me give you one example:

Reed College at that time offered perhaps the best calligraphy instruction in the country. Throughout the campus every poster, every label on every drawer, was beautifully hand calligraphed. Because I had dropped out and didn't have to

take the normal classes, I decided to take a calligraphy class to learn how to do this. I learned about serif and san serif typefaces, about varying the amount of space between different letter combinations, about what makes great typography great. It was beautiful, historical, artistically subtle in a way that science can't capture, and I found it fascinating.

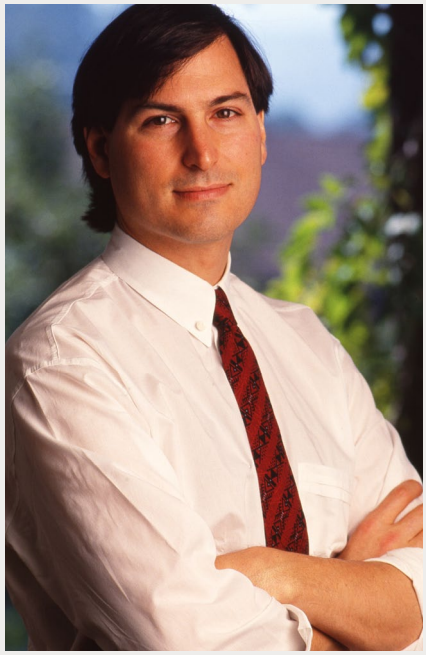
None of this had even a hope of any practical application in my life. But ten years later, when we were designing the first Macintosh computer, it all came back to me. And we designed it all into the Mac. It was the first computer with beautiful typography. If I had never dropped in on that single course in college, the Mac would have never had multiple typefaces or proportionally spaced fonts. And since Windows just copied the Mac, it's likely that no personal computer would have them. If I had never dropped out, I would have never dropped in on this calligraphy class, and personal computers might not have the wonderful typography that they do. Of course it was impossible to connect the dots looking forward when I was in college. But it was very, very clear looking backwards ten years later.

Again, you can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something — your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.

My second story is about love and loss.

I was lucky — I found what I loved to do early in life. Woz and I started Apple in my parents garage when I was 20. We worked hard, and in 10 years Apple had grown from just the two of us in a garage into a $2 billion company with over 4000 employees. We had just released our finest creation — the Macintosh — a year earlier, and I had just turned 30. And then I got fired. How can you get fired from a company you started? Well, as Apple grew we hired someone who I thought was very talented to run the company with me, and for the first year or so things went well. But then our visions of the future began to diverge and eventually we had a falling out. When we did, our Board of Directors sided with him. So at 30 I was out. And very publicly out. What had been the focus of my entire adult life was gone, and it was devastating.

I really didn't know what to do for a few months. I felt that I had let the previous generation of entrepreneurs down — that I had dropped the baton as it was being passed to me. I met with David Packard and Bob Noyce and tried to apologize for screwing up so badly. I was a very public failure, and I even thought about running away from the valley. But something slowly began to dawn on me — I still loved what I did. The turn of events at Apple had not changed that one bit. I had been rejected, but I was still in love. And so I decided to start over.



I didn't see it then, but it turned out that getting fired from Apple was the best thing that could have ever happened to me. The heaviness of being successful was replaced by the lightness of being a beginner again, less sure about everything. It freed me to enter one of the most creative periods of my life.

During the next five years, I started a company named NeXT, another company named Pixar, and fell in love with an amazing woman who would become my wife. Pixar went on to create the worlds first computer animated feature film, Toy Story, and is now the most successful animation studio in the world. In a remarkable turn of events, Apple bought NeXT, I returned to Apple, and the technology we developed at NeXT is at the heart of Apple's current renaissance. And Laurene and I have a wonderful family together.

I'm pretty sure none of this would have happened if I hadn't been fired from Apple. It was awful tasting medicine, but I guess the patient needed it. Sometimes life hits you in the head with a brick. Don't lose faith. I'm convinced that the only thing that kept me going was that I loved what I did. You've got to find what you love. And that is as true for your work as it is for your lovers. Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work. And the only way to do great work is to love what you do. If you haven't found it yet, keep looking.

Don't settle. As with all matters of the heart, you'll know when you find it. And, like any great relationship, it just gets better and better as the years roll on. So keep looking until you find it. Don't settle.

My third story is about death.

When I was 17, I read a quote that went something like: "If you live each day as if it was your last, someday you'll most certainly be right." It made an impression on me, and since then, for the past 33 years, I have looked in the mirror every morning and asked myself: "If today were the last day of my life, would I want to do what I am about to do today?" And whenever the answer has been "No" for too many days in a row, I know I need to change something.

Remembering that I'll be dead soon is the most important tool I've ever encountered to help me make the big choices in life. Because almost everything — all external expectations, all pride, all fear of embarrassment or failure — these things just fall away in the face of death, leaving only what is truly important. Remembering that you are going to die is the best way I know to avoid the trap of thinking you have something to lose. You are already naked. There is no reason not to follow your heart.

About a year ago I was diagnosed with cancer. I had a scan at 7:30 in the morning, and it clearly showed a tumor on my pancreas. I didn't even know what a pancreas was. The doctors told me this was almost certainly a type of cancer that is incurable, and that I should expect to live no longer than three to six months. My doctor advised me to go home and get my affairs in order, which is doctor's code for prepare to die. It means to try to tell your kids everything you thought you'd have the next 10 years to tell them in just a few months. It means to make sure everything is buttoned up so that it will be as easy as possible for your family. It means to say your goodbyes.

I lived with that diagnosis all day. Later that evening I had a biopsy, where they stuck an endoscope down my throat, through my stomach and into my intestines, put a needle into my pancreas and got a few cells from the tumor. I was sedated, but my wife, who was there, told me that when they viewed the cells under a microscope the doctors started crying because it turned out to be a very rare form of pancreatic cancer that is curable with surgery. I had the surgery and I'm fine now.

This was the closest I've been to facing death, and I hope it's the closest I get for a few more decades. Having lived through it, I can now say this to you with a bit more certainty than when death was a useful but purely intellectual concept:

No one wants to die. Even people who want to go to heaven don't want to die to get there. And yet death is the destination we all share. No one has ever escaped it. And that is as it should be, because Death is very likely the single best invention of Life. It is Life's change agent. It clears out the old to make way for the new. Right now the new is you, but someday not too long from now, you will gradually become the old and be cleared away. Sorry to be so dramatic, but it is quite true.

“Have the courage to follow your heart and intuition. They somehow already know what you truly want to become.”

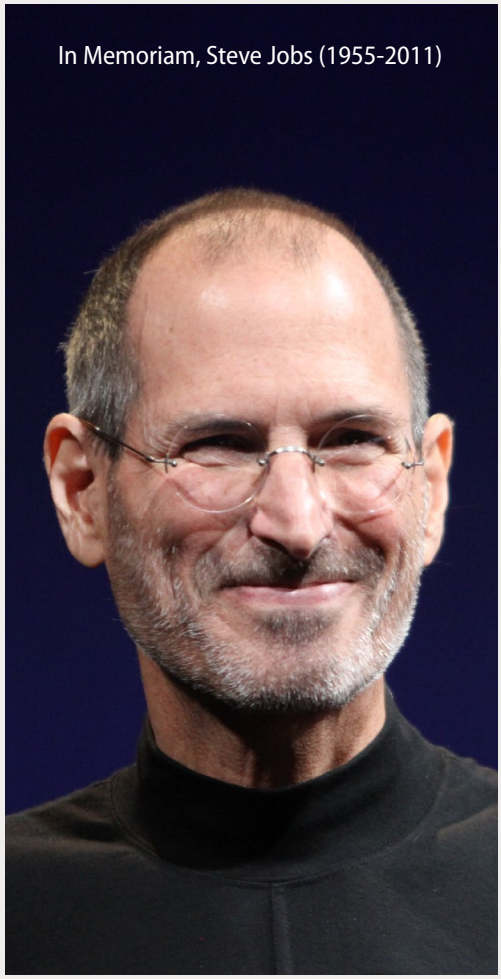
Your time is limited, so don’t waste it living someone else’s life. Don’t be trapped by dogma — which is living with the results of other people’s thinking. Don’t let the noise of others’ opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary.

When I was young, there was an amazing publication called The Whole Earth Catalog, which was one of the bibles of my generation. It was created by a fellow named Stewart Brand not far from here in Menlo Park, and he brought it to life with his poetic touch. This was in the late 1960’s, before personal computers and desktop publishing, so it was all made with typewriters, scissors, and polaroid cameras. It was sort of like Google in paperback form, 35 years before Google came along: it was idealistic, and overflowing with neat tools and great notions.

Stewart and his team put out several issues of The Whole Earth Catalog, and then when it had run its course, they put out a final issue. It was the mid-1970s, and I was your age. On the back cover of their final issue was a photograph of an early morning country road, the kind you might find yourself hitchhiking on if you were so adventurous. Beneath it were the words: “Stay Hungry. Stay Foolish.” It was their farewell message as they signed off. Stay Hungry. Stay Foolish. And I have always wished that for myself. And now, as you graduate to begin anew, I wish that for you.

Stay Hungry. Stay Foolish.

Thank you all very much. ■



In Memoriam, Steve Jobs (1955-2011)

Special thanks to Stanford University for granting the permission to reprint.

Photo Credit (in order of appearance): Robert Holmgren, Joi Ito, Matthew Yohe.

Understanding JIT Spray

By CHRIS LEARY

STEEL YOUR MIND for a tale of intrigue, intertwined with a complex topic in browser security. (It's kind of all over the place, but I might spray something useful.)

Our story, like so many others, starts out with a browser user like yourself, a bottle of red wine, and a devoted young hacker from the Eastern Bloc that answers to the handle "Coleslaw."

Winey-and-Cheesy Corporation, the largest international wine and cheese distributor, has just blitzkrieg bopped the mainstream media over the head with a tactical PR campaign — a free case of wine and sizable wheel of Gouda for the five millionth visitor to their website.

The only problem is that Winey-and-Cheesy's massively trafficked website... has been owned.

Coleslaw is something of a wunderkind, and has, through feats of social engineering and technical prowess paralleled only by terrible movies from the mid 90s, gained the ability to insert some arbitrary, special-sauce HTML and JavaScript into that promotional page.

Coleslaw intends to perform a "zero-day attack" — this means that there's a bug in the browser that Coleslaw knows about, but that the browser vendors are unaware of. Coleslaw thinks that this bug can be used to take over the machines of some unsuspecting users who visit the promotional page, capitalizing on their maniacal love of fine dining.

The Attacker's Dilemma

So, to recap, Coleslaw has found a bug in the browser. Coleslaw wants to exploit that bug in order to obtain *arbitrary code execution* — the ability to run whatever code Coleslaw feels like on the machine that's running the vulnerable browser. The question is, how does Coleslaw get from point A, "I see that there's a bug," to point B, "I can run anything I want on the vulnerable machine"? The process of figuring this out is called *exploit development*.

The exploit development process is a narrative about *control*. Coleslaw starts off by having control over a small set of things — the JavaScript and HTML on a page that the browser visits — but wants to end up controlling everything that the user controls on the vulnerable machine. The environment that internet sites have access to is supposed to be sandboxed; i.e. internet sites are expected to have a somewhat limited and carefully selected set of things it can control. For example, websites that you happen to stumble across shouldn't be able to delete files off of your hard drive.

Strongly-related to this narrative about control is the concept of *determinism*. If Coleslaw has a concrete understanding that performing some action, like calling `alert` from JavaScript, always results in some consequence, like creating and displaying an alert window in the browser, then Coleslaw has effectively extended the realm of control from JavaScript to

triggering-the-code-in-the browser-that-displays-an-alert-dialog. Barring bugs, the realm of control is always confined to the sandbox — the set of possible actions are those that the browser vendor permits an untrusted website to take.

Not All Bugs Are Created Equal

There are lots of different kinds of bugs that browser software can have. There's a relatively tiny set of bugs that permit *control flow hijacking*, which are generally of interest for gaining arbitrary code execution. Successful hijacking implies that you have the ability to control the address of the instruction being executed, which is commonly referred to as pseudo-register `%eip` (where `ip` is short for *instruction pointer*). With full control of `%eip`, the attacker can point it at any executable code — possibly at executable code that they've created.

Control flow hijacking is typically accomplished through some kind of memory corruption, stemming from errors in the use of type-unsafe programming constructs in the browser. In general, the bugs of interest for control flow hijacking are:

- Memory writes that can be used to clobber vtable pointer or function pointer values. The attacker may have control over the location of the memory write, the value being written, or both.
- Buffer overruns that can be used to manipulate values that are ultimately used to determine code to run. The classic example of this is clobbering return addresses present on the C stack.

There's also the possibility of using an attacker-controllable invalid memory read bug to cause an invalid write to happen further along in program execution. Bugs that cause segfaults are carefully evaluated by browser security teams to see if the invalid memory access being performed can be manipulated for use in control flow hijacking.

Platform-level Mitigations: DEP, ASLR, and Canaries

There are some nifty platform-level protections against traditional control flow hijacking techniques. They make both taking control of `%eip` and executing an attacker-controlled code sequence more difficult.

One control-flow hijacking mitigation is stack smashing protection, which is enabled at compile time using a technique referred to as "canary values." An attacker could historically use stack buffer overruns to clobber the return address in a function frame with a target `%eip` value, and the `ret` instruction at the end of the function's machine code would return to that new (attacker-controlled) address value. With this mitigation enabled, however, the compiler places a special value on the stack between local variables (where the buffer lives) and the return value. The compiler also augments the function body with pre-return function prologue code that checks the canary value on the stack against its original value. If a stack buffer overrun causes the return value to be overwritten, the canary that lives in the contiguous space between the locals and return value should indicate that things have gone horribly wrong.

Generally, we tend to think of executables as containing all their executable code as static machine-code. Other than the code that the compiler spat out as specific sections of the executable, nothing else should run over the course of the program's execution. This expectation is codified in an OS-level mitigation called Data Execution Prevention (DEP).

The goal of DEP is to prevent things which are not code from being executed as code at runtime. Your program stack, for example, is just a bunch of space for data that your C function frames can't keep in registers. There's basically no reason that a sane program would ever want to start executing the stack area of memory like it were code. If something like that were to happen, it would be better if your program just terminated, because it could be the pivotal point before an attacker like Coleslaw takes control. Program termination means loss of control for the attacker.

Trying to execute code that was not in the original binary will generally cause the program to fault. In a JIT, however, we purposefully create code at runtime, violating the all-the-code-is-in-the-binary assumption. As a result, we have to explicitly mark the machine code that we create as executable by calling to an operating system API function, like `VirtualProtect` or `mprotect`, to indicate that the data the process has created should really be executable.

DEP's close friend from acronym club is Address Space Layout Randomization (ASLR). ASLR reduces determinism in the process that the attacker is trying to exploit by randomizing the stack address, library loading address, heap address, and PEB address, amongst other key program components. With this mitigation, hardcoded constant addresses in attacker-crafted code become probabilistically unlikely to succeed at hitting their target. As an example, the start of the program stack could wind up being placed at one of 16,000 locations!

This also means that the address of system DLLs, like the ones containing OS API functions like `VirtualProtect` and C library functions like `system`, are probabilistically unknown to the attacker. Since the browser ships linked with all ASLR-enabled DLLs, it's difficult to use linked DLL code as direct footholds in process space.

Coleslaw wants to run an attacker-controlled code payload, but DEP makes it difficult to execute that payload, since it won't be marked as executable by default.

Coleslaw wants to be able to turn the bug that relinquishes control of `%eip` into a reliable exploit, but ASLR makes it difficult to know where to point `%eip` in order to run exploit code.

I imagine that turning a crash into an exploit isn't trivial these days.

Staged Shellcode Payloads

The machine-code payloads that attackers create are referred to as shellcode. Shellcode is generally characterized by its size and its goal, which is usually reflected by the "stage" it's said to be running. For example, the very first shellcode to run, in computer science style, is referred to as "stage 0."

Intermediate stages of shellcode are often used to bootstrap more complex executable code sequences. The complexity involved in turning a bug into an exploit often prevents arbitrarily complex code sequences from executing immediately, so tinier code sequences are written that just delegate responsibility to a more easily formed executable payload. Constraints that apply to the code that the exploit starts running directly tend to disappear after you've gone through some amount of indirection.

Shellcode can easily embed astoundingly small code sequences called "egg hunters" to find the memory address of other attacker-controlled payloads. The egg hunters are designed to avoid faulting the application, because faults cause the attacker to lose control. They work by performing a series of fast-and-minimally-sized system calls to determine whether a virtual memory page is safe to traverse through and read to find the "egg" payload delimiter.

Once the address of a stage 1 data payload is determined, stage 0 shellcode may attempt to make that segment of memory executable. Despite ASLR, the address of the `VirtualProtect` function can be derived by hopping from the known TEB address to the PEB address to the DLL loader address mapping table. Once executable permissions have been added to the stage 1 shellcode, it can simply be jumped to.

Another alternative, if the stage 0 shellcode is executing out of a code space with both writable and executable permissions and sufficient available space, is to use what's called a "GetPC" shellcode sequence to determine the current value of `%eip` and then copy the contents of a stage 1 shellcode payload buffer into the current code space.

For some bugs it may be possible to create "common" stage 0 shellcode to bootstrap any other shellcode payload. This common shellcode is a valuable commodity for exploit toolkits.

JIT Spray, Deconstructed

As mentioned earlier, the JIT has to mark its own assembly buffers as executable. An attacker may look at using that fact to generate executable stage 0 shellcode in order to bypass some of the pain inflicted by DEP. But how could you possibly use JIT compilation process to make shellcode?

> *JIT spraying is the process of coercing the JIT engine to write many executable pages with embedded shellcode.*
> — Blazakis, 2010

Dion Blazakis wrote the seminal paper on JIT spray, in which he presented a jaw-dropping example.

Blazakis noticed that the following ActionScript code:

```
var y = (
    0x3c54d0d9 ^
    0x3c909058 ^
    0x3c59f46a ^
    0x3c90c801 ^
    0x3c9030d9 ^
    0x3c53535b ^
    ...
)
```

Was JIT-compiled into the following instruction sequence:

```
addr    op imm          assembly
0       B8 D9D0543C     MOV EAX,3C54D0D9
5       35 5890903C     XOR EAX,3C909058
10      35 6AF4593C     XOR EAX,3C59F46A
15      35 01C8903C     XOR EAX,3C90C801
20      35 D930903C     XOR EAX,3C9030D9
25      35 5B53533C     XOR EAX,3C53535B
```

Check out the first line. It's showing that the first instruction is a `MOV` that places the 32-bit immediate payload into the `EAX` register. The 32-bit immediate payload from that instruction (`3C54D0D9`) is exactly the immediate that was used as the left-hand-side to the long XOR sequence in the original ActionScript code.

Now, if we look at the subsequent lines, we see that the `addr` column, which is showing the address of instructions relative to the start of the sequence, goes up by `5` every time. That's because each instruction after the initial `MOV` is performing an `XOR` against the original value in the accumulator register, `EAX`, exactly as the ActionScript program described.

Each of these instructions is exactly 5 bytes long — each instruction has a 1-byte opcode prefix, given under the `op` column, followed by a 32-bit immediate constant: the opcode for `MOV EAX,[imm32]` is `0xB8`, and the opcode sequence for `XOR EAX,[imm32]` is `0x35`.

The immediate column may look confusing at a glance, but it's actually just the little-endian equivalent of the 32-bit immediate given in the assembly: the "little end" (least significant byte) goes "in" (at the lowest memory address), which is why the byte order looks flipped around from the one given in the assembly (and in the original ActionScript program).

It may not look so sinister, but the above table is actually deceiving you! In the table, all of the instructions are the same number of bytes (5) in length. On x86 CPUs, however, instructions are actually a variable number of bytes in length: instructions can be as small as a single byte, but can get quite long: the `nop` instruction is just a 0x90 opcode byte with no operands, whereas the `movl $0xdeadbeef, 0x12345678(%ebx,%edx,1)` instruction is significantly larger.

As a result, when we look at this instruction sequence "crooked" (with a 1-byte skew to the address), we decode a totally different sequence of instructions. I'll show you what I mean.

Our instructions in memory look like the following buffer:

```
static const char buf[] = {
    0xB8, 0xD9, 0xD0, 0x54, 0x3C,
    0x35, 0x58, 0x90, 0x90, 0x3C,
    0x35, 0x6A, 0xF4, 0x59, 0x3C,
    0x35, 0x01, 0xC8, 0x90, 0x3C,
    0x35, 0xD9, 0x30, 0x90, 0x3C,
    0x35, 0x5B, 0x53, 0x53, 0x3C
};
```

When we load this up in GDB, and run the `disassemble` command, we confirm the instructions present in the above table:

```
(gdb) disassemble/r buf
Dump of assembler code for function buf:
   0x08048460 <+0>:  b8 d9 d0 54 3c mov    eax,0x3c54d0d9
   0x08048465 <+5>:  35 58 90 90 3c xor    eax,0x3c909058
   0x0804846a <+10>: 35 6a f4 59 3c xor    eax,0x3c59f46a
   0x0804846f <+15>: 35 01 c8 90 3c xor    eax,0x3c90c801
   0x08048474 <+20>: 35 d9 30 90 3c xor    eax,0x3c9030d9
   0x08048479 <+25>: 35 5b 53 53 3c xor    eax,0x3c53535b
```

But then, if we look at the buffer with a 1-byte offset, we see a totally different set of instructions! Note the use of `buf+1` as the disassembly target:

```
(gdb) disassemble/r (buf+1), (buf+sizeof(buf))
Dump of assembler code from 0x8048461 to 0x804847e:
   0x08048461 <buf+1>:      d9 d0  fnop
   0x08048463 <buf+3>:      54     push   esp
   0x08048464 <buf+4>:      3c 35  cmp    al,0x35
   0x08048466 <buf+6>:      58     pop    eax
   0x08048467 <buf+7>:      90     nop
   0x08048468 <buf+8>:      90     nop
   0x08048469 <buf+9>:      3c 35  cmp    al,0x35
   0x0804846b <buf+11>:     6a f4  push   0xfffffff4
   0x0804846d <buf+13>:     59     pop    ecx
   0x0804846e <buf+14>:     3c 35  cmp    al,0x35
   0x08048470 <buf+16>:     01 c8  add    eax,ecx
   0x08048472 <buf+18>:     90     nop
   0x08048473 <buf+19>:     3c 35  cmp    al,0x35
   0x08048475 <buf+21>:     d9 30  fnstenv [eax]
   0x08048477 <buf+23>:     90     nop
   0x08048478 <buf+24>:     3c 35  cmp    al,0x35
   0x0804847a <buf+26>:     5b     pop    ebx
   0x0804847b <buf+27>:     53     push   ebx
```

If you look down the middle part of the two disassemblies, before the assembly mnemonics, you can read that the bytes are the same from left to right: the first line of the first disassemblies goes `b8 d9 d0 54 3c`, and the second disassembly starts on the second byte of that same sequence with `d9 d0 54 3c`, straddling multiple instructions. This is the magic of variable length instruction encoding: when you look at an instruction stream a little bit sideways, things can change very drastically.

Yo Dawg, I Heard You Like X86 Assembly...

It's not obvious, at first glance, just how clever this technique is.

The goal of the ActionScript code pattern is for the attacker to insert arbitrary bytes into the code stream that the JIT otherwise generates. The attacker then uses these arbitrary bytes as an alternate instruction stream. However, the attacker has to compensate for the non-attacker-controlled bytes that surround its own.

Each 32-bit immediate encoded in the ActionScript program starts with a MSB of `0x3c`. That byte is little-endian encoded and placed, in memory, right before each of the `0x35`s that represent the `XOR EAX,[imm32]` opcode.

Jumping to the 1-byte offset from the base address of the instruction stream starts us off executing `0xd9 0xd0`, a 2-byte instruction that runs a no-op on the floating point unit. Both of these bytes were part of the attacker's immediate value: `0x3c54d0d9`.

Effectively, the attacker is able to control 4 out of every 5 bytes per instruction in the stream. They are somewhat limited by the bytes fixed in the instruction stream, however. The MSB of each immediate is a `0x3c` so that it can successfully combine with the `0x35` from the `XOR EAX,[imm32]` opcode to create a nop-like instruction, `cmp al,0x35`, that keeps the stream executing at the 1-byte offset.

It would be ideal for the attacker if they could find a way to incorporate the `0x35` into an instruction in a useful way, instead of having to lose a byte in order to control it; however, there are lots of fun tricks that you can play to make compact instruction sequences. By making use of the stacky subset of x86 you can get a nice little MISCy program: pushes and pops are nice 1-byte instructions that you can split across the semantic nops to simulate moves, and pushing 8-bit signed immediates only takes 2 bytes, as you can see at `buf+11`. Dumping your floating point coprocessor state out to the stack is a 2-byte sequence. Accessing the TEB is a 3-byte sequence. How can you not love x86?

For this particular code sequence, the attacker only has a 1 in 5 chance of jumping to an `%eip` that gives control back to the JIT program. If you land anywhere in the constant-encoded portion, the instruction sequence will be entirely different.

Outstanding Issues

So now we know the basic requirements for pulling off a JIT spray attack:

- Deterministic attacker control of values embedded in the instruction stream
- Control of `%eip`
- The ability to jump somewhere inside the JIT code, in order to probabilistically execute the attacker's interleaved instruction stream

But wait, how do you know *where* to jump?

JIT spray opens up the possibility for an attacker to create a lot of very similar code via the JIT compiler, possibly with nop sled prefixes. As a result, one approach to bypassing both DEP and ASLR is to fill enough of the address space with JIT code that you can jump to a random location and hit an attacker-controlled portion valid JIT code buffer with reasonable probability.

But this leads to further questions: what address does the attacker pick to jump to? How much code memory does the attacker spray? Creating a reliable exploit seems significantly more difficult.

Blazakis' Solution

In order to create a reliable exploit (as opposed to a probabilistic one), Blazakis used the techniques of *pointer inferencing* and *heap feng shui*.

The sandbox makes it particularly tricky to figure out where things live in memory. Those kinds of details definitely aren't supposed to be exposed through the sandbox. If the attacker were able to figure out the locations of things in memory space through the sandbox, it would be considered an information leak.

Pointer inferencing is the technique that Blazakis used to accurately determine the memory location of heapified ActionScript entities in the Flash VM. The inferencing described in Blazakis' paper is cleverly based on the fact that literal integer values in the Flash virtual machines are hashed alongside of heap-object pointers. By observing the default dictionary enumeration order — the order in which keys exist in the hash table — Blazakis was able to narrow down the value of the object pointer to its exact location.

"Heap feng shui" is the process of understanding the memory allocation behaviors of the sandboxed environment that code is running in, and using that knowledge to place objects in some known locations in memory relative to each other. Blazakis noted that the ActionScript object heap expands in 16MiB increments and took into account the heuristics for executable allocations when loading ActionScript bytecode entities. Blazakis also relied on the usage of `VirtualAlloc` in the ActionScript memory allocator, with the knowledge that `VirtualAlloc` maps the first 64KiB aligned hole that's found in a linear scan through the virtual address space.

Blazakis was able to combine these techniques into reliable stage 0 shellcode execution by:

1. Determining the exact pointer of the first object within a 16MiB heap chunk.
2. Spraying just enough JIT code to place a JIT code allocation right after that 16MiB chunk.
3. Determining the JIT spray address to be the object address + 16MiB.
4. Adding a value like 0x101 to the base address to get an **unaligned JIT code location**, as described in the JIT spray section above.
5. Jumping to that resulting address.

Back to the Story: the Law of Large Numbers

So, Coleslaw intends to use a multi-step process:

1. Find bug that permits control flow hijacking
2. Perform JIT spray
3. Jump to probabilistic address for stage 0 shellcode

Importance of leaked information about the memory map becomes apparent here: it prevents you from doing a JIT-spray and jump-spray. However, given enough visitors, like the 5 million to Winey-and-Cheesy's giveaway, we have to start calculating expected values. As mitigations are added to lower the probability of success, we can see the *expected value of ownage* drop as well. ■

Chris Leary is a Mozilla JavaScript engine hacker working on JIT spray mitigations for an upcoming version of Firefox.

Reprinted with permission of the original author.
First appeared in *hn.my/jit* (cdleary.com)

Evolutionary Algorithm

Evolving "Hello, World!"

By FERRY BOENDER

My interest in Evolutionary Algorithms started when I read On the Origin of Circuits over at DamnInteresting.com. I always wanted to try something like that out for myself, but never really found the time. Now I have, and I think I've found some interesting results.

Disclaimer: I know next to nothing about Evolutionary Algorithms. Everything you read in here is the product of my own imagination and tests. I may use the wrong algorithms, nomenclature, and methodology, and I might just be getting very bad results. They are, however, interesting to me, and I do know something about evolution, so here it is anyway.

How Evolution Works

So, how does an Evolutionary Algorithm work? Why, the same as normal biological evolution, mostly! Very (very) simply said, organisms consist of DNA, which determines their characteristics. When organisms reproduce, there is a chance their offspring's DNA contains a mutation, which can lead to differences in characteristics. Sufficiently negative changes in offspring make that offspring less fit to survive, causing it, and the mutation, to die out eventually. Positive changes are passed on to future offspring. So through evolution, a set of DNA naturally tends to grow towards its "goal," which is ultimate fitness for its environment. Now this is not an entirely correct description, but for our purposes it is good enough.

A Simple Evolutionary Algorithm

There is nothing stopping us from using the same technique to evolve things towards goals set by a programmer. As can be seen from the Antenna example in the DamnInteresting article, this can sometimes even produce better things than engineers can come up with. For example, I'm going to evolve the string "Hello, World!" from random garbage. The first example won't be very interesting, but it demonstrates the concept rather well.

First, let's define our starting point and end goal:

```
source = "jiKnp4bqpmAbp"
target = "Hello, World!"
```

Our evolutionary algorithm will start with "jiKnp4bqpmAbp", which we can view as the DNA of our "organism." It will then randomly mutate some of the DNA, and judge the new mutated string's fitness. But how do we determine fitness? This is probably the most difficult part of any evolutionary algorithm.

Lucky for us, there's an easy way to do this with strings. All we have to do is take the value of each character in the mutated string, and see how much it differs from the same character in the target string. This is called the distance between two characters. We then add all those differences, which leads us to a single value which is the fitness of that string. A fitness of 0 is perfect, and means that both strings are exactly the same. A fitness of 1 means one of the characters is off by one. For instance, the strings "Hfllo" and "Hdllo" both have a fitness of one. The higher the fitness number, the less fit it actually is!

Here's the fitness function.

```
def fitness(source, target):
   fitval = 0
   for i in range(0, len(source)):
      fitval += (ord(target[i]) - ord(source[i])) ** 2
   return(fitval)
```

If you look closely, you'll notice that for each character, I square the difference. This is to convert any negative numbers to positive ones, and to put extra emphasis on larger differences. If we don't do this, the string "Hannp" would have a fitness of 0. You see, the difference between "e" and "a" is -5, between "l" and "n" is +2 (which we have twice) and between "o" and "p" is +1. Adding these up yields a fitness of 0, but it's not the string we want at all. If we square the differences, they become 25, 4, 4 and 1, which yields a fitness of 34. Effectively, we square each difference so that they can't cancel each other out.

Now we need to introduce mutations into our string. This is rather easy. We simply pick a random character in the string, and either increment or decrease it by one, or leave it alone:

```
def mutate(source):
   charpos = random.randint(0, len(source) - 1)
   parts = list(source)
   parts[charpos] = chr(ord(parts[charpos])
+ random.randint(-1,1))
   return(''.join(parts))
```

Time to tie the whole shebang together!

```
fitval = fitness(source, target)
i = 0
while True:
   i += 1
   m = mutate(source)
   fitval_m = fitness(m, target)
   if fitval_m < fitval:
      fitval = fitval_m
      source = m
      print "%5i %5i %14s" %
(i, fitval_m, m)
   if fitval == 0:
      break
```

This should be easy enough to understand. For each iteration of the While-loop, we mutate the string and then calculate its fitness. If it is fitter then the original string (the parent), we make the child the new string. Otherwise, we throw it away. If the fitness is 0, we're done!

Let's look at some output. I'm snipping out some intermediary output because it's not terribly interesting.

At generation 1, we have a fitness of 15491, and the string looks nothing like "Hello, World!" The same for generation 20, 40, 60, etc.

```
  1 15491  jjKnp4bqpmAbp
 20 15400  jiKnp3bppoAbp
 40 15377  jiKlo2bpooAdp
 60 15130  iiKlo2aoooAdp
```

Not much progress so far. At generation 500 it's still a load of nonsense:

```
500 9986 \eTlo,YaorNdf
```

Generation 1200: we start to see something that looks like "Hello, World!":

```
1200 4186 Heglo,LWorhdP
```

Generation 1500: we're getting very close!

```
1500  3370  Hello,GWorldL
```

It still takes a good 1500 generations more before we're finally there:

```
3078     2  Hello, Vorld"
3079     2  Hflo, World"
3080     2  Hflo, World"
3081     0  Hello, World!
```

There it is!

A Better, More Interesting, Algorithm

Okay, so that worked. But...it was kind of lame. Nothing interesting to see, really, was there? That's because our algorithm was a little too simplistic. Only one "organism" in the gene pool, only one character mutated at any time. We can do better than that, so let's modify the program to make it more interesting.

We're not going to touch our fitness function, since that works rather well. Instead, let's introduce a gene pool. Instead of having only one string, why not have a whole bunch of randomly generated strings and let them duke it out among themselves. That sounds a bit more real-life, doesn't it?

```
GENSIZE = 20
genepool = []
for i in range(0, GENSIZE):
   dna = [random.choice(string.printable[:-5]) for j
in range(0, len(target))]
   fitness = calc_fitness(dna, target)
   candidate = {'dna': dna, 'fitness': fitness }
   genepool.append(candidate)
```

This little snippet generates a gene pool with 20 random strings and their fitnesses. In an official implementation, the gene pool would be called the **population**.

Now, let's modify our mutation function. Instead of mutating one single character, we feed it two parents, picked at random from the genepool, and it will mix their DNA together a bit. This is called "**crossover**". It will also randomly mutate one character in the resulting DNA. It then returns the newly fabricated child, including its fitness.

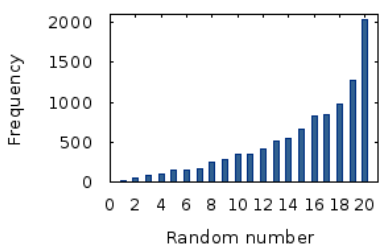
```
def mutate(parent1, parent2):
   child_dna = parent1['dna'][:]

   # Mix both DNAs
   start = random.randint(0, len(parent2['dna']) - 1)
   stop = random.randint(0, len(parent2['dna']) - 1)
   if start > stop:
      stop, start = start, stop
   child_dna[start:stop] = parent2['dna'][start:stop]

   # Mutate one position
   charpos = random.randint(0, len(child_dna) - 1)
   child_dna[charpos] = chr(ord(child_dna[charpos]) +
random.randint(-1,1))
   child_fitness = calc_fitness(child_dna, target)
   return({'dna': child_dna, 'fitness': child_fitness})
```

We also need a routine to pick two random parents from the genepool. Now, we could just pick them completely random, but what you really want is for parents with a good fitness to have a better chance of offspring. This is called "**elitism**." If we sort the genepool list by fitness, we can use a uniform product distribution to make sure that parents with better fitness get chosen more often.

Now you might ask, what the hell is a uniform product distribution? When you randomly pick a number between, say, 1 and 10, each number has the same chance of being picked. This is called a "uniform distribution." But when you pick two random numbers, and you multiply them, there's a much bigger chance of getting a bigger number than a smaller number. Hence the name "uniform product distribution." Here's how that looks:



So our random parent picker will do just that. We select two random real numbers between 0 and 1, multiple those two random numbers and then scale the result up to our pool size by multiplying the result with the size of the pool. We return that parent from the pool.

```
def random_parent(genepool):
   wRndNr = random.random() * random.random() * (GENSIZE - 1)
   wRndNr = int(wRndNr)
   return(genepool[wRndNr])
```

There! Now it's time for our main loop.

```
while True:
   genepool.sort(key=lambda candidate: candidate['fitness'])

   if genepool[0]['fitness'] == 0:
      # Target reached
      break

   parent1 = random_parent(genepool)
   parent2 = random_parent(genepool)

   child = mutate(parent1, parent2)
   if child['fitness'] < genepool[-1]['fitness']:
      genepool[-1] = child
```

For each iteration of the While True loop, we first sort the genepool by fitness so that the most fit parents are at the top. We check to see if the fittest happens to be the target string we're looking for. If so, we stop the loop.

Then we select two parents from the genepool using the uniform product distribution so that fitter parents are chosen more often. We create a bastard mutated child that will mix both parents' DNA together and introduce a little mutation. If the new child is more fit than the worst in the genepool, it will replace that degenerate one in the genepool. In the next iteration, the pool is sorted again on fitness so that the new child takes its rightful place.

Results

Now it's time to run this puppy and see what it does. Again, I snip out some of the less interesting stuff.

Here's the genepool in the beginning. The first number is the generation (the number of times the While-loop has run), the second number is the fitness, and the third column is the DNA for that entry in the genepool.

```
1   7617   'iSx{$,K`u~(B
1   9284   SQf`1N#UdrPlT
1  12837   sYIu<E"Fq'^_.
1  15531   DC8Dg1I$*mUs-
1  16064   L~*}JBVdF7bu2
1  16533   1,XU%)5$q[YuO
1  16588   ff],ceW<0fud&
1  17316   [V3@2'VgY\{KV
1  17356   kWw#v/P<#apG9
1  17581   <Lrh(1hN_Bd)3
1  18777   TM]_]TbtxFY:q
1  19656   $zS+EI?BS>%z(
1  19841   =S;B~((W8 D,6
1  20398   P_A$D|NPJPio/
1  21957   J&f=O:g\8'{S2
1  22543   5*T2c"pMZ80L'
1  24954   A&lZ#A_}MxI"P
1  25186   &9MrI|0&x)q,N
1  28110   OlXT/Q{y3{"LR
1  29656   8WB99hx%0]}h[
```

One big random jumbled mess. Note the ones I've emphasized. These are the parents that were selected for the new child in the next generation. Let's see how it looks after one generation:

```
2   7617   'iSx{$,K`u~(B
2   8742   SQf`1N#UdfumT
2   9284   SQf`1N#UdrPlT
2  12837   sYIu<E"Fq'^_.
2  15531   DC8Dg1I$*mUs-
2  16064   L~*}JBVdF7bu2
2  16533   1,XU%)5$q[YuO
2  16588   ff],ceW<0fud&
2  17316   [V3@2'VgY\{KV
2  17356   kWw#v/P>#apG9
2  17581   <Lrh(1hN_Bd)3
2  18777   TM]_]TbtxFY:q
2  19656   $zS+EI?BS>%z(
2  19841   =S;B~((W8 D,6
2  20398   P_A$D|NPJPio/
2  21957   J&f=O:g\8'{S2
2  22543   5*T2c"pMZ80L'
2  24954   A&lZ#A_}MxI"P
2  25186   &9MrI|0&x)q,N
2  28110   OlXT/Q{y3{"LR
```

Two random parents from the previous generation have their DNA mixed, and have generated an offspring (the bold one) which is better then both of them. It comes in second with a fitness of 8742, while its parents only had fitness of 9284 and 16588. Let's skip ahead a bit and look at the 6th generation:

```
6   7617   'iSx{$,K`u~(B
6   8742   SQf`1N#UdfumT
6   9284   SQf`1N#UdrPlT
6  10198   SQfD1N#UdfumT
6  12837   sYIu<E"Fq'^_.
6  15531   DC8Dg1I$*mUs-
6  16064   L~*}JBVdF7bu2
6  16387   SQf`1N"MZ80LT
6  16533   1,XU%)5$q[YuO
6  16588   ff],ceW<0fud&
6  17316   [V3@2'VgY\{KV
6  17356   kWw#v/P>#apG9
6  17356   kWw#v/P>#apG9
6  17581   <Lrh(1hN_Bd)3
6  18777   TM]_]TbtxFY:q
6  19656   $zS+EI?BS>%z(
6  19841   =S;B~((W8 D,6
6  20287   fe],1eW<0fud&
6  20398   P_A$D|NPJPio/
6  21957   J&f=O:g\8'{S2
```

As you can see, the "SQf" has reproduced again with success, and there are now four variants of it in the genepool. We also note the "kWw#", of which there are two identical ones. This can happen when the entire DNA of one parent is copied and no mutation occurs. In our mutate function, we use the first parent's DNA as a base and then randomly overlay some of the seconds parent's DNA. This can be anything from the entire second parent's DNA, or nothing at all. But generally, the chance is higher that the first parent's DNA survives largely in tact.

The next interesting generation is 13:

```
13   4204   RQf`{$,KdfumT
13   7617   'iSx{$,K`u~(B
13   7617   'iSx{$,K`u~(B
13   8742   SQf`1N#UdfumT
13   8742   SQf`1N#UdfumT
13   9284   SQf`1N#UdrPlT
13   9284   SQf`1N#UdrPlT
13  10198   SQfD1N#UdfumT
13  12837   sYIu<E"Fq'^_.
13  15531   DC8Dg1I$*mUs-
13  15838   L~*xJBVdG7bu2
13  15856   $zS+<E"Fq(^_(
13  15883   L~*xJCVdG7bu2
13  16064   L~*}JBVdF7bu2
13  16387   SQf`1N"MZ80LT
13  16533   1,XU%)5$q[YuO
13  16588   ff],ceW<0fud&
13  17316   [V3@2'VgY\{KV
13  17356   kWw#v/P>#apG9
13  17356   kWw#v/P>#apG9
```

Wow! "SQf" has been really busy and now almost rules the genepool. "iSx" is second and third, but has lost its number one position to the "RQf" variant of "SQf." "RQf" was introduced in the 12th generation as a child of an "iSx" and "SQf" variant. We see that "kWv" has been knocked almost to the end of the list by more fit candidates. It is very obvious that this pool is no longer random. Patterns are starting to emerge all over it.

By the time we reach generation 40:

```
40   3306   RQSw{$-KcfumB
40   4204   RQf`{$,KdfumT
40   4229   RQf`|$,KdfumT
40   4242   RQe`|$,KdfumT
40   4795   RQSw{$-KdfumT
40   4971   RQSwz$*K`uSnT
40   4973   RQSwz$+K`uSmT
40   4992   RQSwz$+K`uSnT
40   5017   SQSxz$+K`uSmT
40   5017   SQSxz$+K`uSmT
40   5951   (QSxz$+KdfSmT
40   5985   'QSxz$+K`uSmT
40   6421   SQfx{$+K`u~(B
40   6444   TQf`{$+K`u~(B
40   6489   SQfx{$+KdfS(B
40   6492   TQf`{$-K`u~(B
40   7034   SQSxy$+KdfS(B
40   7617   'iSx{$,K`u~(B
40   7617   'iSx{$,K`u~(B
40   7625   'iS`{$,Kdg~(B
```

The genepool is now almost entirely dominated by the "RQf" variants. Forms of its original parents "SQf" and "iSx" can still be found here and there, although "iSx" is almost entirely gone from the pool. An interesting thing is that we can see combinations of letters (bold) that keep reappearing. These are almost like actual genes! Combinations of DNA that work well together and therefore stay in the genepool in that combination. It takes lots of generations to make variants of these genes that are more fit then previous versions.

The next milestone is found in the 67th generation:

```
67   3138   RQSw{$+KdfukA
67   3161   RQSw{$+KcfukA
67   3176   RQSw{$,KdfulA
67   3176   RQSw{$+KcfulA
67   3218   RQSw{$-LcfumA
67   3222   RQSw{%,KefumB
67   3237   RQSw{$-LcfvmA
67   3241   RQSw{$-KcfumA
67   3241   RQSw{$-KcfumA
67   3266   RQSw{$-KceumA
67   3266   RQSw{$-KceumA
67   3267   RRSw{$-KcfumB
67   3289   RQSw{%,KefumC
67   3306   RQSw{$-KcfumB
67   3306   RQSw{$-KcfumB
67   3323   RQSw{#-KcfumB
67   3324   RPSw{$-KdfumB
67   3331   RQSw{$-KbfumB
67   3348   RQSw{#-KbfumB
67   3489   RQSw{$+KdfumA
```

This marks the first generation where there are no other variations than the RQS one. But immediately, we see the next generation in which a new number one is found:

```
68   3119   QQSw{$+KdfukA
68   3138   RQSw{$+KdfukA
68   3161   RQSw{$+KcfukA
```

By the 96th generation, QQS has taken over the top:

```
96   3060   QQSw{%+KdhukA
96   3065   QRSw{%+KdfukA
96   3081   QQSw{%+KdgukA
96   3081   QQSw{%+KdgukA
96   3081   QQSw{%+KdgukA
96   3096   QQSw{$+KdgukA
96   3104   QQSw{%+KdfukA
96   3119   QQSw{$+KdfukA
96   3119   QQSw{$+KdfukA
96   3119   QQSw{$+KdfukA
96   3137   RRSw{$,KdfulA
96   3137   RRSw{$,KdfulA
96   3138   RQSw{$+KdfukA
96   3138   RQSw{$+KdfukA
96   3138   RQSw{$+KdfukA
96   3138   RQSw{$+KdfukA
96   3138   RQSw{$+KdfukA
96   3142   QQSw{$,KdfukA
96   3142   QQSw{$+KcfukA
96   3144   QQSw|$+KdfukA
```

This is where the race gets boring. Every now and then a new, better, mutation will arise and take over the genepool. Change is slow, though, and no big surprises are left. The candidates slowly but surely mutate until they reach something resembling the "Hello, World!" we are looking for in generation 1600:

```
1600    19   Hdllo+ Worle%
1600    20   Hdklo+ Worle%
1600    20   Hdklo+ Worle%
1600    20   Hdklo+ Worle%
1600    20   Hdklo+ Worle%
1600    20   Hdklo+ Workd%
```

It takes almost another half-thousand generation to get to the final target:

```
1904     0   Hello, World!
1904     1   Hello, World"
1904     1   Hello, World"
1904     2   Hello, Wprld"
1904     2   Helmo, World"
1904     2   Helmo, World"
1904     2   Hdllo, World"
1904     2   Hello, Worle"
```

Interesting facts:

- It usually takes anywhere between 2500 and 4000 generations to evolve the target.
- On average, it takes approximately 3100 generations to evolve the target.
- If we remove the parent DNA mixing and rely solely on mutations, it takes on average 3650 generations to evolve the target.
- The parent DNA mixing is only really useful in the beginning. In the first generations, it can quickly propel a new mix of DNA to the top of the list, but later on random mutations instead of mixing DNA becomes the main driving force between the evolutions. (This doesn't have to be the case in real life evolution, naturally.) ■

Ferry Boender is a Software Engineer and hacker with over 20 years of programming experience. He holds a Bachelor's degree in Computer Science.

Reprinted with permission of the original author.
First appeared in *hn.my/evolution* (electricmonk.nl)

Bash Shortcuts For Maximum Productivity

By ALAN SKORKIN

IT MAY OR may not surprise you to know that **the *bash* shell has a very rich array of convenient shortcuts** that can make your life, working with the command line, a whole lot easier. This ability to edit the command line using shortcuts is provided by the GNU Readline library. This library is used by many other **nix* applications besides *bash*, so learning some of these shortcuts will not only allow you to zip around bash commands with absurd ease, but also make you more proficient in using a variety of other **nix* applications that use Readline. I don't want to get into Readline too deeply so I'll just mention one more thing. By default Readline uses *emacs* key bindings. Although it can be configured to use the *vi* editing mode, I prefer to learn the default behavior of most applications (I find it makes my life easier not having to constantly customize stuff). If you're familiar with *emacs* then many of these shortcuts will not be new to you. These are mostly for the rest of us.

Command Editing Shortcuts

- `Ctrl + a` → go to the start of the command line
- `Ctrl + e` → go to the end of the command line
- `Ctrl + k` → delete from cursor to the end of the command line
- `Ctrl + u` → delete from cursor to the start of the command line
- `Ctrl + w` → delete from cursor to start of word (i.e., delete backwards one word)
- `Ctrl + y` → paste word or text that was cut using one of the deletion shortcuts (such as the one above) after the cursor
- `Ctrl` + xx →move between start of command line and current cursor position (and back again)
- `Alt + b` → move backward one word (or go to start of word the cursor is currently on)
- `Alt + f` → move forward one word (or go to end of word the cursor is currently on)
- `Alt + d` → delete to end of word starting at cursor (whole word if cursor is at the beginning of word)
- `Alt + c` → capitalize to end of word starting at cursor (whole word if cursor is at the beginning of word)
- `Alt + u` → make uppercase from cursor to end of word
- `Alt + l` → make lowercase from cursor to end of word
- `Alt + t` → swap current word with previous
- `Ctrl + f` → move forward one character
- `Ctrl + b` → move backward one character
- `Ctrl + d` → delete character under the cursor
- `Ctrl + h` → delete character before the cursor
- `Ctrl + t` → swap character under cursor with the previous one

Command Recall Shortcuts

- `Ctrl + r` → search the history backwards
- `Ctrl + g` → escape from history searching mode
- `Ctrl + p` → previous command in history (i.e., walk back through the command history)
- `Ctrl + n` → next command in history (i.e., walk forward through the command history)
- `Alt + .` → use the last word of the previous command Command Control Shortcuts
- `Ctrl + l` → clear the screen
- `Ctrl + s` → stops the output to the screen (for long-running verbose command)
- `Ctrl + q` → allow output to the screen (if previously stopped using command above)
- `Ctrl + c` → terminate the command
- `Ctrl + z` → suspend/stop the command

Bash Bang (!) Commands

Bash also has some handy features that use the ! (bang) to allow you to do some funky stuff with bash commands.

- `!!` → run last command
- `!blah` → run the most recent command that starts with "blah" (e.g., !ls)
- `!blah:p` → print out the command that !blah would run (also adds it as the latest command in the command history)
- `!$` → the last word of the previous command (same as Alt + .)
- `!$:p` → print out the word that !$ would substitute
- `!*` → the previous command except for the last word (e.g., if you type 'find some_file.txt /', then !* would give you 'find some_file.txt')
- `!*:p` → print out what !* would substitute

There is one more handy thing you can do. This involves using the ^^ "command." If you type a command and run it, you can re-run the same command but substitute a piece of text for another piece of text using ^^. For example:

```
$ ls -al
total 12
drwxrwxrwx+ 3 Admin None    0 Jul 21 23:38 .
drwxrwxrwx+ 3 Admin None    0 Jul 21 23:34 ..
-rwxr-xr-x  1 Admin None 1150 Jul 21 23:34 .bash_profile
-rwxr-xr-x  1 Admin None 3116 Jul 21 23:34 .bashrc
drwxr-xr-x+ 4 Admin None    0 Jul 21 23:39 .gem
-rwxr-xr-x  1 Admin None 1461 Jul 21 23:34 .inputrc
$ ^-al^-lash
ls -lash
total 12K
   0 drwxrwxrwx+ 3 Admin None    0 Jul 21 23:38 .
   0 drwxrwxrwx+ 3 Admin None    0 Jul 21 23:34 ..
4.0K -rwxr-xr-x  1 Admin None 1.2K Jul 21 23:34 .bash_profile
4.0K -rwxr-xr-x  1 Admin None 3.1K Jul 21 23:34 .bashrc
   0 drwxr-xr-x+ 4 Admin None    0 Jul 21 23:39 .gem
4.0K -rwxr-xr-x  1 Admin None 1.5K Jul 21 23:34 .inputrc
```

Here, the command was the `^-al^-lash`, which replaced the `-al` with `-lash` in our previous `ls` command and re-ran the command again.

There is a lot more that you can do when it comes to using shortcuts with bash. But, the shortcuts above will get you 90% of the way towards maximum bash productivity. ■

Alan Skorkin is a developer and aspiring software craftsman from Melbourne, Australia. He is often found causing controversy on his blog skorks.com, while sharing his thoughts about hacking, the software development profession and the people who work in it.

Reprinted with permission of the original author. First appeared in *hn.my/bash* (skorks.com)

Finding Love Optimally

By MICHAEL TRICK

LIKE MANY IN operations research, my research interests often creep over into my everyday life. Since I work on scheduling issues, I get particularly concerned with everyday scheduling, to the consternation of my friends and family ("We should have left 6 minutes ago: transportation is now on the critical path!"). This was particularly true when I was a doctoral student when, by academic design, I was living and breathing operations research 24 hours a day.

I was a doctoral student from ages 22 to 27, and like many in that age group, I was quite concerned with finding a partner with whom to spend the rest of my life. Having decided on certain parameters for such a partner (female, breathing, etc.), I began to think about how I should optimally find a wife. In one of my classes, it hit me that the problem has been studied: it is the Secretary Problem! I had a position to fill (secretary, wife, what's the difference?), a series of applicants, and my goal was to pick the best applicant for the position.

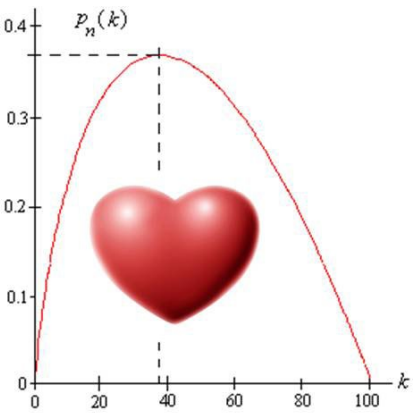
Fortunately, there is quite a literature on the Secretary Problem, and there are a number of surprising results. The most surprising is that it is possible to find the best secretary with any reasonable probability at all. The hard part is that each candidate is considered one at a time, and an immediate decision must be made to accept or reject the candidate. You can't go back and say

Heart Of The Storm, *flickr.com/photos/jdhancock/4354438814*

"You know, I think you are the cat's meow after all." This matched up with my empirical experience in dating. Further, at each step, you only know if the current candidate is the best of the ones you have seen: candidates do not come either with objective values or with certifications of perfection, again matching empirical observations. You can only compare them with what you have sampled.

Despite these handicaps, if you know how many candidates there are, there is a simple rule to maximize the chance of finding the best mate: sample the first K candidates without selecting any of them, and then take the first subsequent candidate who is the best of all you have seen. K depends on N, the total number of candidates you will see. As N gets big, K moves toward 1/e times N, where e is 2.71….So sample 36.9% of the candidates, and then take the first candidate who is the best you have seen. This gives a 36.9% chance of ending up with Ms. or Mrs. Right.

One problem here: I didn't know what N is. How many eligible women will I meet? Fortunately, the next class covered that topic. If you don't know what N is but know that you will be doing this over a finite amount of time T, then it is okay to replace this with a time cutoff rule: simply take the first candidate after 36.9% of the time (technically, you use 36.9% of the cumulative distribution, but I assumed a uniform distribution of candidate arrivals). Okay, I figured, people are generally useless at 40 (so I thought then: the 50-year-old-me would like to argue with that assumption), and start this matching process at about 18 (some seem to start earlier, but they may be playing a different game), so, taking 36.9% of the 22 year gap gives an age of 26.11. That was my age! By a great coincidence, operations research had taught me what to do at exactly the time I needed to do that.



Redoubling my efforts, I proceeded to sample the candidate pool (recognizing the odds were against me: there is still only a 36.9% chance of finding Ms. Right) when lo and behold — I met her: the woman who was better than every previous candidate. I didn't know if she was Perfect (the assumptions of the model don't allow me to determine that), but there was no doubt that she met the qualifications for this step of the algorithm. So I proposed.

And she turned me down.

And that is when I realized why it is called the Secretary Problem, and not the Fiancée Problem (though Merrill Flood proposed the problem under that name). Secretaries have applied for a job and, presumably, will take the job if offered. Potential mates, on the other hand, are also trying to determine their best match through their own Secretary Problem. In order for Ms. Right to choose me, I had to be Mr. Right to her! And then things get much more complicated. What if I was meeting women in their sampling phase? It did seem that some people were very enthusiastic about having long sampling phases, and none of them would be accepting me, no matter how good a match they would be for me. And even the cutoff of 36.9% looks wrong in this case. In order to have a hope of matching up at all in this "Dual Secretary Problem," it looked like I should have had a much earlier cutoff, and in fact, it seemed unlikely there was a good rule at all!

I was chagrined that operations research did not help me solve my matching problem. I had made one of the big mistakes of practical operations research: I did not carefully examine the assumptions of my model to determine applicability.

Downcast, I graduated with my doctorate, resolving to marry myself to integer programming. I embarked on a postdoc to Germany.

There, I walked into a bar, fell in love with a beautiful woman, moved in with her 3 weeks later, invited her to live in the United States "for a while," married her 6 years after that, and had a beautiful son with her 6 years ago. I am not sure what optimization model led me down that path, but I think I am very happy with the result. ■

Michael Trick is a professor at Carnegie Mellon, who really did meet his wife as a postdoc in Germany. His research is in mathematical optimization, with an emphasis on applications in voting and in sports.

Reprinted with permission of the original author. First appeared in *hn.my/love* (mat.tepper.cmu.edu)

Things I Learned On A Round-The-World Yacht Race

By TONY HAILE

ELEVEN YEARS AGO this month, I stepped aboard a 72-foot racing cutter affectionately called The Good Ship Logica and began a 10-month round the world yacht race, the only one to go around the world against the currents and prevailing winds. Below deck, I was the geek, making sure the satellite could broadcast despite 90ft waves blocking line of sight; above deck I was the bowman, standing at the pointy end and getting the shit kicked out of me by walls of water as our team struggled to take down huge sails that the wind wanted to keep up.

Today I learned that someone mishandled a crane in Portsmouth during a routine maneuver and dropped Logica, effectively killing it. This was the boat that I learned to trust to keep me safe through hurricanes, lightning strikes and the worst the Southern Ocean had to offer. It was the boat that I cursed every time a rampant wave picked me up and tossed me down the deck like a rag doll, slamming me into rigging and stanchions. It was the boat in whose bowels I spent cold hours pumping water into buckets after the electric pump failed, the boat that taught me how to sleep on a rollercoaster while a generator roared next to my head, the boat I loved, heart and soul. Now she's gone.

So today I've been thinking about the lessons she taught me.

The Opposite of Fear Is Not Bravery, It's Initiative

When my first hurricane at sea hit, it came out of nowhere. I was delivering a boat (the older, smaller sister of Logica) across the Atlantic from Plymouth to Boston. The boom swung across the deck with such ferocity that it ripped the pulley system that controlled it out of the deck and flung it out to sea; the third wave took the heavily bolted down compass and consigned that to the ocean. Our skipper was up on deck so fast it seemed incredible that he had just been asleep and, screaming above the waves, he got us working to try to bring down the mainsail and control the wayward boom. Our boat was so far over on its side that the mast was dipping into the ocean and water was starting to drag the mainsail and the boat further down into the lifeless grey. I don't remember being frightened, at least not in the way I had always thought about fear; traditional fear involves some prediction of a future you would rather avoid. At this point, I couldn't begin to think about a future at all. I just remember feeling utterly drained of initiative. I would do whatever anyone asked me to do, but I was utterly unable to think or to act for myself.

I brooded over that night for months afterwards, dwelling on my own inadequate response when faced with a true crisis. I knew I was due to set out on a round-the-world yacht race the next year and was terrified that I didn't have what it takes, that I would let down my team when it mattered most.

In October 2000, my skipper came below decks and asked us if we had ever seen the Perfect Storm (it had occurred on the Grand Banks near our position at the time). "Yeah, three storms converging on the Flemish Cap," replied Adam, the bowman on the other watch. "We're in luck," the skipper replied, "we've only got two storms converging on us." We watched the scarlet dawn rising and remarked upon the sailors motto "red sky at night, sailor's delight; red sky at morning, we're fucked."

We had more warning this time, but the hurricane still hit with a vengeance. There's something about the sea when the wind gets above 70 knots of breeze (80mph), it becomes gunmetal grey, as if not even color could live in these conditions. Our bow team struggled up to the foredeck to take down the headsails and put up our storm staysail. Orange and bulletproof, we needed it up if we were going to be able to steer a course through this storm at all. This was the moment I had thought about for years, but for some reason I was not the same man who had been so useless on that previous voyage. I was able to think, to act on my own initiative and help my team to survive. It was a revelation and gave me hope that the ability to lead in a crisis was not inbuilt from birth but could be learned, that I could become better. The lesson I took from this is that bravery is a term applied retroactively, after the work has been done and the danger has passed. In a situation that engenders fear and terror, don't ask yourself to be brave; simply ask yourself to act. The bravery comes later.

Finding Fault is a Luxury Best Saved for Tomorrow

My first day of training on the yacht, and I'd already managed to break something. A sail was tumbling down and the boat was losing speed. The first mate darted across the boat to find out what had happened and I started in on a long and rambling tale of the series of unfortunate events which had, through no fault of my own, caused the damage we were looking at right now. I was barely three sentences in, when the mate interrupted me: "I don't give a crap whose fault it was, I just need to know what to fix."

The words hit me like a sledgehammer, my concern had been with my perceived reputation and standing as a competent crewman, his concern was simply that the boat wasn't working right and yet it needed to be. Identifying the

incompetent culprit responsible or working out the precise series of events leading us to here were luxuries that could wait for another time. Right now the boat needed to be fixed before we lost too much speed and time. If I was ever going to truly pull my weight with the crew, I would have to learn to be ok with people potentially thinking the worst of me or ascribing failures to me that were not directly my own fault, what mattered was keeping the boat moving. I find thinking of that day instructive when facing a board meeting. Finding fault or assigning blame is an idle luxury — what matters is keeping the company moving.

Do Your Thinking Before The Crisis

We were deep in the Southern Ocean, one of the nastiest environments on earth, and three of us were sitting on the windward side of the deck (the high side) with little to do but endure the waves crashing over us and make sure the helmsman didn't get hurt. Our skipper came up on deck to take a look around and spotted a trailing rope on the leeward side that he wanted to tidy. He made his way down to where the deck was skimming the water and began to bring in the rope when a rogue wave took him by surprise and knocked him down the deck. All three of us leaped forward to grab him before he was washed overboard, but two of us were stopped short by our safety lines like a dog reaching the limits of its leash.

Only Glyn, had the presence of mind to first unhook his safety line get across to the other side, reattach and reach our skipper before it was too late. While I and my team-mate had been sitting there grumpily bearing the waves and wishing we were elsewhere, Glyn had been running through scenarios in his head and working out potential plans of action should any of them occur. He knew that there isn't necessarily time in a crisis to stop, assess the best course of action and then enact it, so you have to do your thinking beforehand. Be constantly working through "what if?" scenarios so that your brain has the advantage when an accident happens and you are not left flailing helplessly at the end of a line watching someone get washed away.

Leave It on the Last Wave

Our round the world yacht race involved putting 18 people in a tin can, plunging it in salt water, and shaking it violently for 10 months. People hallucinate through lack of sleep, the unconscious tapping of teeth can provoke a knife fight (which occurred on another yacht in a previous race), and one simply can't avoid someone if you have an argument. The only way for your team to mentally survive in that kind of environment is to embody the motto of "leave it on the last wave." The argument you had during a sail change? That happened on a wave way in the distance, leave it out there where it belongs. The time you almost came to blows with a team mate over something so minor you both can't remember, leave it on the wave where it started because the wind has changed and there are new sails to be put up and a new course to take. The lesson on a boat is clear: you can either let go of slights or negative emotions or you can damn near kill someone. There's not much wiggle room in between.

These are some of the gifts that Logica gave me. My friends have often remarked upon how the person who joined the race in September 2000 was utterly different from the man who left it in July of 2001. I miss my boat, I miss my team, and I will always treasure what I learned on her deck. ■

Tony Haile is CEO of Chartbeat and an all-round troublemaker at Betaworks. Prior to his life in startups, Tony competed in a round-the-world yacht race, was Editor of the Middle East and international terrorism desk for Control Risks and managed to get paid to muck about on polar expeditions. He has stood at the North Pole, worked sail changes under the Southern Lights and married a Pennsylvanian.

Reprinted with permission of the original author.
First appeared in *hn.my/yacht* (tonyhaile.com)

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Please contact promo@magcloud.com with any questions.

MAGCLOUD