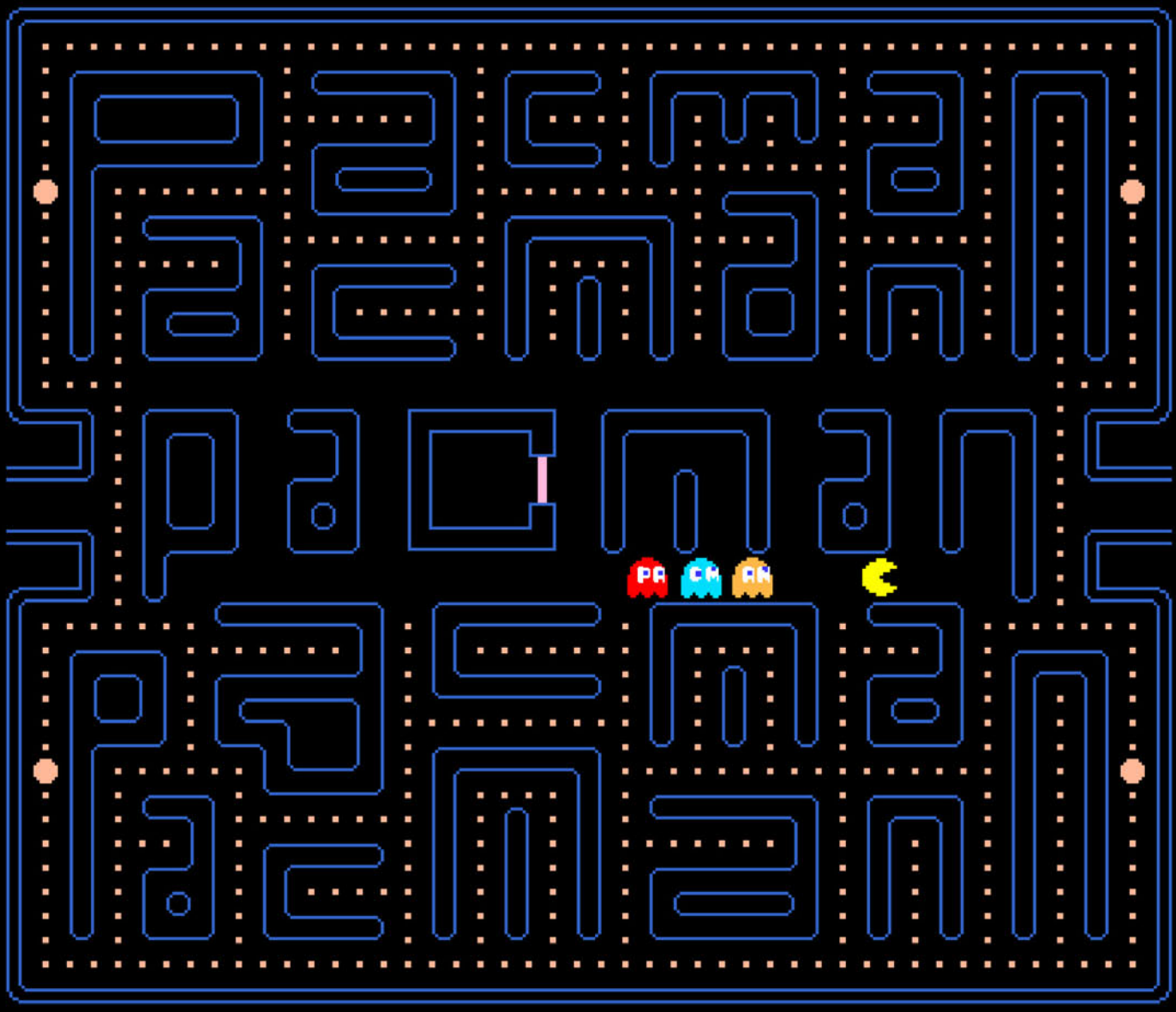


Understanding Pac-man Ghost Behaviour

HACKERMONTHLY Issue 10 March 2011

Curator
Lim Cheng Soon

Proofreader
Jordan Greenaway

Printer
MagCloud

Contributors

ARTICLES
Daniel Tenner
Chad Birch
Martin Kleppmann
Rahul Vohra
Gabriel Weinberg
Evan Miller
Mike Saunders
Kenneth Ballenegger
James Hague
Cody Brocious

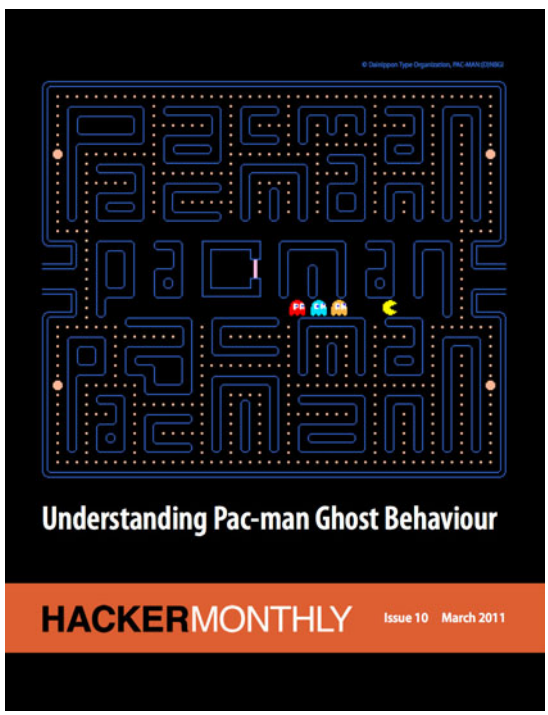
COMMENTARIES
Ed Weissman
Michael Melanson
Jacques Mattheij

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 Art Work: Dainippon Type Organization

Contents

For links to the posts on Hacker News, visit *hackermonthly.com/issue-10*. All articles and comments are reprinted with permission of their original author.
Hacker Monthly is published by Netizens Media and not affiliated with Y Combinator in any way.

How To Get Your Startup On Hacker News

By DANIEL TENNER

In networking events, surprisingly often I get asked about how to get a startup to be discussed, somehow, on Hacker News [news.ycombinator.com].

Why get on Hacker News?

Getting your startup on Hacker News (HN) is useful:

- HN is a thriving community of entrepreneurs — probably the biggest on the web — and their feedback is valuable. It's often thorough, honest, qualified, and sometimes unpleasant. It can help you improve your early startup significantly: better define your offering, tune your landing page, etc.
- There are many early tech adopters on HN, so if the app is useful to them, you can get some early users.
- Because HN is made of entrepreneurs, they're not allergic to commercial offerings like many other popular forums on the web. They won't get offended if you're trying to make money from your hard work. In fact, they'll probably suggest better business models for extracting money from your users.
- Finally, coverage on HN can sometimes lead to being picked up by other tech websites. I know a few editors of major tech news sites who browse HN regularly.

So, on the whole, I think getting your early stage startup reviewed by the HN community is a no-brainer. You can only gain from it.

How should you go about it?

The best thing to do is surprisingly simple: post an "Ask HN: Review my startup, xyz.com" post.

To do this, submit a post with no URL following the pattern above. Then, connect to the #startups channel on the Freenode IRC network and ask people for their opinion. If your request for feedback seems legitimate, you will probably get some comments and some upvotes, and that will lead, eventually, to more attention from the wider HN community.

Some tips about this process:

- Do this when you're genuinely looking for feedback. Posting an "Ask HN: Review my startup, xyz.com" when your company has been going for 5 years and has 20 employees would be disingenuous (and probably useless because, by then, you probably know more about how to run your business than we do).
- The number of actual comments can vary greatly depending on the time of day, the day of the week, the other stories in the community, the startup that you present, etc. The number of responses will easily vary from a handful to over a hundred. Don't take it personally.
- Be open and responsive. When people post feedback, don't get defensive, or they won't provide any more feedback. Accept it, and try to respond constructively, maybe asking a few questions for clarification. Never tell the person providing feedback that they're wrong.

- Ask for specific feedback on areas where you have concerns (e.g., "I'm particularly looking for feedback about the UI.").
- Include the link in the body of your post. It won't be hotlinked, but don't worry about that. Do not include it as the "URL" part of your post, some people consider that bad form.
- Don't ask HN to review your startup several times within a short span of time. It's okay to ask for more feedback a few months later, but refer back to your earlier post and mention how you addressed the earlier feedback.
- If you don't get any upvotes within the first half hour or so, it may well happen that no-one sees your post, and particularly if you submitted it at a very active time (e.g., midday Eastern Standard Time). Make sure you get some people, such as the IRC channel, to have a look within the first half an hour.

You don't strictly have to follow these rules. In fact, the top "Ask HN: Review ..." posts on SearchYC [searchyc.com] have no post-body and have a URL, but those tend to be exceptions. If you want to maximise your chances of getting feedback, even if your startup is not yet great, try to do something like these examples: *hn.my/review1*, *hn.my/review2*.

Also, Paul Graham mentioned that "Review my startup" posts submitted by brand new users will often be deleted as spam. If you want to submit this kind of post, it's a good idea to create a user some time before then, and be reasonably active for at least a few weeks before posting about your startup.

What if you're not looking for feedback?

Some startups are past the initial feedback stage. That's fine. They can still get on HN, but the approach is different. If you want to get some hackers's attention at a later stage in your startup, the best way, in my experience, is to share something valuable with the community. For example:

- A post detailing how you addressed a certain technological challenge with plenty of meaty details. For example, "How we scaled our video encoding from 10 daily users to 10,000 without buying more hardware."
- A post with an original viewpoint about some aspect of starting and running a startup. Unoriginal points tend not to get upvoted, so try to find a unique slant. For example, "Why we chose to have no permanent employees."
- A post that "opens the kimono" — talking about how your sales have gone, what worked, what didn't, and so on. All in a way that readers can learn from. For example, "Our sales for the first 12 months."
- A post that tells a very personal story about some aspect of your startup of the kind that people normally don't hear about. For example, "How we dealt with a co-founder dropping out in the first 6 months of the startup."

There are many other types of posts that work. If you read HN with some regularity, you'll start to recognize them. ■

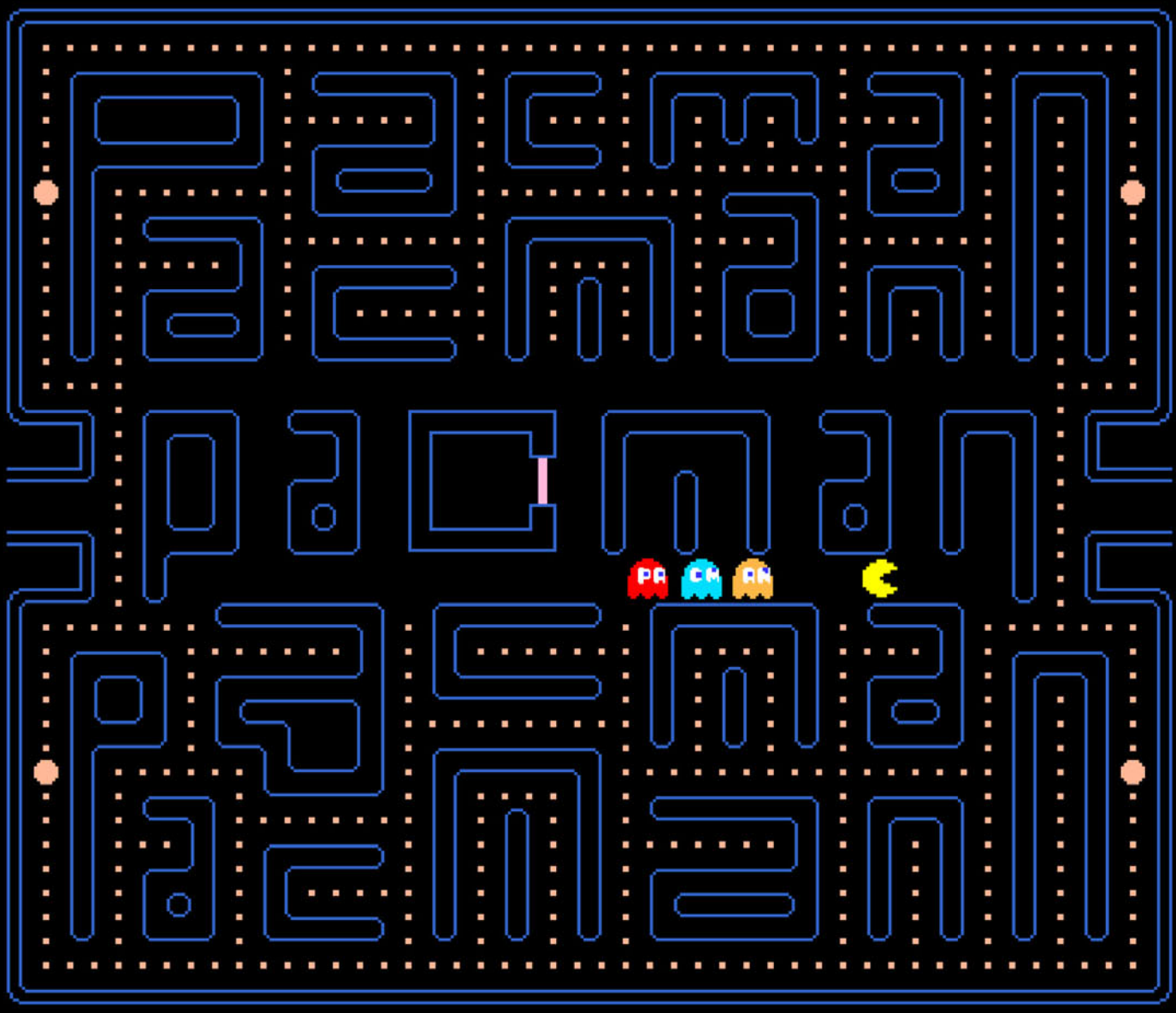
Daniel Tenner is the founder of Woobius [www.woobius.com] and GrantTree [www.granttree.co.uk]. Known as "swombat" on Hacker News and Twitter, he has been blogging about startups for three years now, since he started his first company, and is now producing *swombat.com* a daily updated resource for people who like to read startup articles like this one.

Reprinted with permission of the original author. First appeared in *hn.my/startuphn.*

Commentary

By ED WEISSMAN (edw519)

The most critical words are genuinely and disingenuous. You may be able to get something over on us every once in a while, but for the most part, this is a crowd that doesn't respond too well to posers and B.S. Be sincere and we'll tell you the truth. Be phony and we'll probably send you packing.



Understanding Pac-Man Ghost Behavior

By CHAD BIRCH

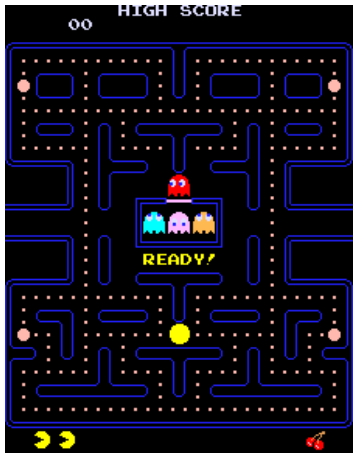
It only seems right for me to begin this article with the topic that inspired me to start it in the first place. Not too long ago, I came across Jamey Pittman's "Pac-Man Dossier [hn.my/dossier]," which is a ridiculously detailed explanation of the mechanics of Pac-Man. I found it absolutely fascinating, so this site is my attempt to discover and aggregate similarly detailed information about other games. But, as a bit of a tribute, I'm going to start with Pac-Man as well, specifically the ghost AI. It's an interesting topic, and hopefully my explanation will be a bit more accessible than Jamey's due to focusing on only the information relevant to ghost behavior.

About the Game

> *"All the computer games available at the time were of the violent type — war games and space invader types. There were no games that everyone could enjoy, and especially none for women. I wanted to come up with a "comical" game women could enjoy."*
> – Toru Iwatani, Pac-Man creator

Pac-Man is one of the most iconic video games of all time, and most people (even non-gamers) have at least a passing familiarity with it. The purpose of the game is very simple: the player is placed in a maze filled with food (depicted as pellets or dots) and needs to eat all of it to advance to the next level. This task is made difficult by four ghosts that pursue Pac-Man through the maze. If Pac-Man makes contact with any of the ghosts, the player loses a life and the positions of Pac-Man and the ghosts are reset, although any dots that were eaten remain so. Other than simply avoiding them, Pac-Man's only defense against the ghosts are the four larger "energizer" pellets located at the corners of the maze. Eating one causes the ghosts to become frightened and retreat for a short time, and in the early levels of the game Pac-Man can even eat the ghosts for bonus points during this period. An eaten ghost is not completely eliminated, but is returned to its starting position before resuming its pursuit. Other than eating dots and ghosts, the only other source of points are the two pieces of fruit which appear during each level near the middle of the maze. The first fruit appears when Pac-Man has eaten seventy of the dots in the maze, and the second when 170 have been eaten.

Every level of Pac-Man uses the same maze layout, containing 240 regular "food" dots and four energizers. The tunnels that lead off of the left and right edges of the screen act as shortcuts to the opposite side of the screen, and are usable by both Pac-Man and the ghosts, although the ghosts's speed is greatly reduced while they are in the tunnel. Even though the layout is always the same, the levels become increasingly difficult due to modifications to Pac-Man's speed, as well as changes to both the speed and behavior of the ghosts. After reaching level twenty-one, no further changes to the game's mechanics are made, and every level from twenty-one onwards is effectively identical.



Common Elements of Ghost Behaviour

> *"Well, there's not much entertainment in a game of eating, so we decided to create enemies to inject a little excitement and tension. The player had to fight the enemies to get the food. And each of the enemies has its own character. The enemies are four little ghost-shaped monsters, each of them a different color — blue, yellow, pink, and red. I used four different colors mostly to please the women who play — I thought they would like the pretty colors."*
> – Toru Iwatani, Pac-Man creator

Each of the ghosts is programmed with an individual "personality" — a different algorithm it uses to determine its method of moving through the maze. Understanding how each ghost behaves is extremely important to be able to effectively avoid them. But, before discussing their individual behaviors, let's first examine the logic that they share.

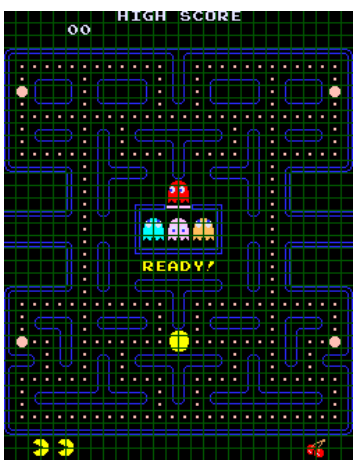
The Ghost House

When a player begins a game of Pac-Man, they are not immediately attacked by all four of the ghosts. As shown on the diagram of the initial game position, only one ghost

begins in the actual maze, while the others are inside a small area in the middle of the maze, often referred to as the "ghost house". Other than at the beginning of a level, the ghosts will only return to this area if they are eaten by an energized Pac-Man, or as a result of their positions being reset when Pac-Man dies. The ghost house is otherwise inaccessible, and is not a valid area for Pac-Man or the ghosts to move into. Ghosts always move to the left as soon as they leave the ghost house, but they may reverse direction almost immediately due to an effect that will be described later.

The conditions that determine when the three ghosts that start inside the ghost house are able to leave it are actually fairly complex. Because of this, I'm going to consider them outside the scope of this article, especially since they become much less relevant after completing the first few levels. If you're interested in reading about these rules (and an interesting exploit of them), Pittman's "Pac-Man Dossier" covers them in-depth, as always.

Target Tiles



Much of Pac-Man's design and mechanics revolve around the idea of the board being split into tiles. "Tile" in this context refers to an 8-by-8 pixel square on the screen. Pac-Man's screen resolution is 224-by-288, so this gives us a total board size of 28-by-36 tiles, although most of these are not accessible to Pac-Man or the ghosts. As an example of the impact of tiles, a ghost is considered to have caught Pac-Man when it occupies the same tile as him. In addition, every pellet in the maze is in the center of its own tile. It should be noted that since the sprites for Pac-Man and the ghosts are larger than one tile in size, they are never completely contained in a single tile. Due to this, for the game's purposes, the character is considered to occupy whichever tile contains its centerpoint. This is important knowledge when avoiding ghosts, since Pac-Man will only be caught if a ghost manages to move its centerpoint into the same tile as Pac-Man's.

The key to understanding ghost behavior is the concept of a target tile. The large majority of the time, each ghost has a specific tile that it's trying to reach, and its behavior revolves around trying to get to that tile from its current one. All of the ghosts use identical methods to travel towards their targets, but the different ghost personalities come about due to the individual way each ghost has of selecting its target tile. Note there are no restrictions that a target tile must actually be possible to reach, they can (and often are) located on an inaccessible tile, and many of the common ghost behaviors are a direct result of this possibility. Target tiles will be discussed in more detail in upcoming sections, but for now just keep in mind that the ghosts are almost always motivated by trying to reach a particular tile.

Ghost Movement Modes

The ghosts are always in one-of-three possible modes: "Chase", "Scatter", or "Frightened". The "normal" mode with the ghosts pursuing Pac-Man is Chase, and this is the one that they spend most of their time in. While in Chase mode, all of the ghosts use Pac-Man's position as a factor in selecting their target tile, although it is more significant to some ghosts than others. In Scatter mode, each ghost has a fixed target tile, each of which is located just outside a different corner of the maze. This causes the four ghosts to disperse to the corners. Frightened mode is unique because the ghosts do not have a specific target tile. Instead, they pseudorandomly decide which turns to make at every intersection. A ghost in Frightened mode also turns dark blue, moves much more slowly and can be eaten by Pac-Man. But, the duration of Frightened mode is shortened as the player progresses through the levels, and is completely eliminated from level nineteen onwards.

> *"To give the game some tension, I wanted the monsters to surround Pac-Man at some stage of the game. But I felt it would be too stressful for a human being like Pac-Man to be continually surrounded and hunted down. So, I created the monsters's invasions to come in waves. They'd attack and then they'd retreat. As time went by they would regroup, attack, and disperse again. It seemed more natural than having constant attack."*
>
> – Toru Iwatani, Pac-Man creator

Changes between Chase and Scatter modes occur on a fixed timer, which causes the "wave" effect described by Iwatani. This timer is reset at the beginning of each level and whenever a life is lost. The timer is also paused while the ghosts are in Frightened mode, which occurs whenever Pac-Man eats an energizer. When Frightened mode ends, the ghosts return to their previous mode, and the timer resumes where it left off. The ghosts start out in Scatter mode, and then there are four waves of Scatter-Chase alternation, after which the ghosts will remain in Chase mode indefinitely (until the timer is reset). For the first level, the durations of these phases are:

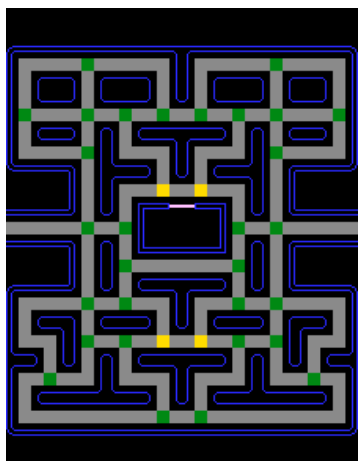
1. Scatter for seven seconds, then Chase for twenty seconds.
2. Scatter for seven seconds, then Chase for twenty seconds.
3. Scatter for five seconds, then Chase for twenty seconds.
4. Scatter for five seconds, then switch to Chase mode permanently.

The durations of these phases are changes somewhat when the player reaches level two, and once again when they reach level five. Starting on level two, the third Chase mode lengthens considerably, to 1033 seconds (seventeen minutes and thirteen seconds), and the following Scatter mode lasts just sixtieth of a second before the ghosts proceed to their permanent Chase mode. The level 5 changes build on top of this, additionally reducing the first two Scatter lengths to 5 seconds, and adding the 4 seconds gained here to the third Chase mode, lengthening it to 1037 seconds (seventeen minutes and seventeen seconds). Regarding the sixtieth of a second Scatter mode on every level except the first, even though it may seem that switching modes for such an insignificant amount of time is pointless, there is a reason behind it, which shall be revealed shortly.

Basic Ghost Movement Rules

The next step is understanding exactly how the ghosts attempt to reach their target tiles. The ghosts's AI is very simple and short-sighted, which makes the complex behavior of the ghosts even more impressive. Ghosts only ever plan one step into the future as they move about the maze. Whenever a ghost enters a new tile, it looks ahead to the next tile that it will reach, and makes a decision about which direction it will turn when it gets there. These decisions have one very important restriction, which is that ghosts may never choose to reverse their direction of travel. That is, a ghost cannot enter a tile from the left side and then decide to reverse direction and move back to the left. The implication of this restriction is that whenever a ghost enters a tile with only two exits, it will always continue in the same direction.

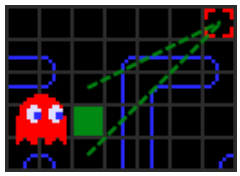
But, there is one exception to this rule, which is that whenever ghosts change from Chase or Scatter to any other mode, they are forced to reverse direction as soon as they enter the next tile. This forced instruction will overwrite whatever decision the ghosts had previously made about the direction to move when they reach that tile. This effectively acts as a notifier to the player that the ghosts have changed modes, since it is the only time a ghost can possibly reverse direction. Note that when the ghosts leave Frightened mode they do not change direction, but this particular switch is already obvious due to the ghosts reverting to their regular colors from the dark blue of Frightened. So then, the sixtieth of a second Scatter mode on every level after the first will cause all the ghosts to reverse their direction of travel, even though their target effectively remains the same. This forced direction reversal is also applied to any ghosts still inside the ghost house, so a ghost that hasn't yet entered the maze by the time the first mode switch occurs will exit the ghost house with a "reverse direction as soon as you can" instruction already pending. This causes them to move left as usual for a very short time, but they will almost immediately reverse direction and go to the right instead.



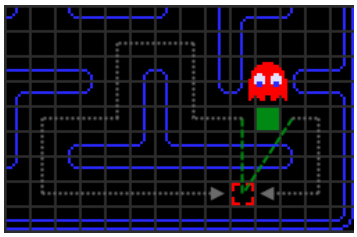
This diagram shows a simplified representation of the maze layout. Decisions are only necessary at all when approaching "intersection" tiles, which are indicated in green on the diagram.

When a decision about which direction to turn is necessary, the choice is made based on which tile adjoining the intersection will put the ghost nearest to its target tile, measured in a straight line. The distance from every possibility to the target tile is

measured, and whichever tile is closest to the target will be selected. In the diagram to the left, the ghost will turn upwards at the intersection. If two or more potential choices are an equal distance from the target, the decision between them is made in the order of up > left > down. A decision to exit right can never be made in a situation where two tiles are equidistant to the target, since any other option has a higher priority.



Since the only consideration is which tile will immediately place the ghost closer to its target, this can result in the ghosts selecting the "wrong" turn when the initial choice places them closer, but the overall path is longer. An example is shown to the right, where the beeline measurement makes exiting left appear to be a better choice. But, this will result in an overall path length of twenty-six tiles to reach the target, while exiting right would have had a path of only eight tiles long.



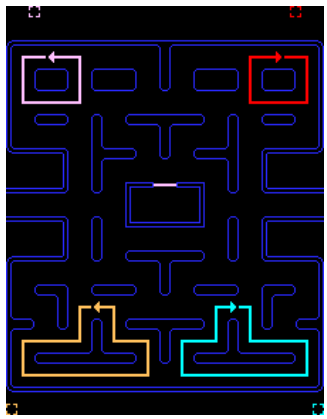
One final special case, the four intersections that were colored yellow on the simplified maze diagram. These specific intersections have an extra restriction — ghosts cannot choose to turn upwards from these tiles. If entering them from the right or left side they will always proceed out the opposite side (excepting a forced direction reversal). Note that this restriction does not apply to Frightened mode and Frightened ghosts may turn upwards here if that decision occurs randomly. A ghost entering these tiles from the top can also reverse direction back out the top if a mode switch occurs as they are entering the tile. The restriction is only applied during "regular" decision making. If Pac-Man is being pursued closely by ghosts, he can gain some ground on them by making an upwards turn in one of these intersections, since they will be forced to take a longer route around.

Individual Ghost Personalities

> *"This is the heart of the game. I wanted each ghostly enemy to have a specific character and its own particular movements, so they weren't all just chasing after Pac-Man in single file, which would have been tiresome and flat."*
> – Toru Iwatani, Pac-Man creator

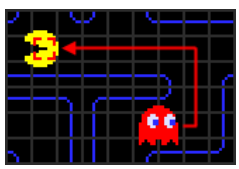
As has been previously mentioned, the only differences between the ghosts are their methods of selecting target tiles in Chase and Scatter modes. The only official description of each ghost's personality comes from the one-word "character" description shown in the game's attract mode. We'll first take a look at how the ghosts behave in Scatter mode, since it's extremely straightforward, and then look at each ghost's approach to targeting in Chase mode.

Scatter Mode



Each ghost has a predefined, fixed target tile while in this mode, located just outside the corners of the maze. When Scatter mode begins, each ghost will head towards their "home" corner using their regular methods. But, since the actual target tiles are inaccessible and the ghosts cannot stop moving or reverse direction, they are forced to continue past the target, but will turn back towards it as soon as possible. This results in each ghost's path eventually becoming a fixed loop in their corner. If left in Scatter mode, each ghost would remain in its loop indefinitely. In practice, the duration of Scatter mode is always quite short, so the ghosts often do not have time to even reach their corner or complete a circuit of their loop before reverting back to Chase mode. The diagram shows each ghost's target tile and eventual looping path, color-coded to match their own color.

The Red Ghost

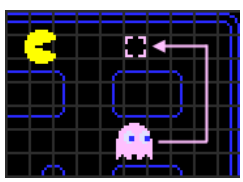


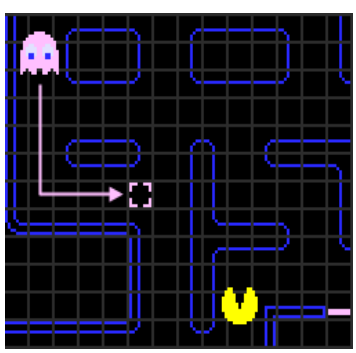
The red ghost starts outside of the ghost house, and is usually the first one to be seen as a threat, since he makes a beeline for Pac-Man almost immediately. He is referred to as Blinky, and the game describes his personality as shadow. In Japanese, his

personality is referred to as 追いかけ, oikake, which translates as "pursuer" or "chaser". Both languages's descriptions are accurate, since Blinky's target tile in Chase mode is defined as Pac-Man's current tile. This ensures that Blinky almost always follows directly behind Pac-Man, unless short-sighted decision making causes him to take an inefficient path.

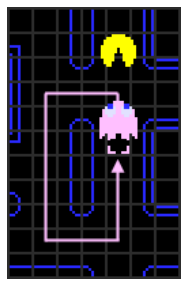
Even though Blinky's targeting method is very simple, he does have one idiosyncrasy that the other ghosts do not: at two defined points in each level (based on the number of dots remaining), his speed increases by 5% and his behavior in Scatter mode changes. The timing of the speed change varies based on the level, with the change occurring earlier and earlier as the player progresses. The change to Scatter targeting is perhaps more significant than the speed increases, since it causes Blinky's target tile to remain as Pac-Man's position even while in Scatter mode, instead of his regular fixed tile in the upper-right corner. This effectively keeps Blinky in Chase mode permanently, though he will still be forced to reverse direction as a result of a mode switch. When in this enhanced state, Blinky is generally referred to as "Cruise Elroy", although the origin of this term seems to be unknown. Not even the almighty "Pac-Man Dossier" has an answer here. If Pac-Man dies while Blinky is in Cruise Elroy mode, he reverts back to normal behavior temporarily, but returns to Elroy mode as soon as all other ghosts have exited the ghost house.

The Pink Ghost



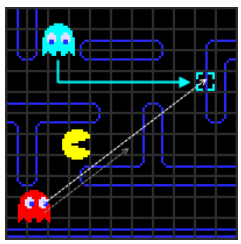


The pink ghost starts inside the ghost house, but always exits immediately, even in the first level. His nickname is Pinky, and his personality is described as speedy. This is a considerable departure from his Japanese personality description, which is 待ち伏せ, machibuse, which translates as "ambusher". The Japanese version is much more appropriate, since Pinky does not move faster than any of the other ghosts (and slower than Blinky in Cruise Elroy mode), but his targeting scheme attempts to move him to the place where Pac-Man is going, instead of where he currently is. Pinky's target tile in Chase mode is determined by looking at Pac-Man's current position and orientation, and selecting the location four tiles straight ahead of Pac-Man. At least, this was the intention, and it works when Pac-Man is facing to the left, down, or right, but when Pac-Man is facing upwards, an overflow error in the game's code causes Pinky's target tile to actually be set as four tiles ahead of Pac-Man and four tiles to the left of him. I don't wish to frighten off non-programmers, but if you're interested in the technical details behind this bug Don Hodges has written a great explanation [hn.my/pinky], including the actual assembly code for Pinky's targeting, as well as a fixed version.



One important implication of Pinky's targeting method is that Pac-Man can often win a game of "chicken" with him. Since his target tile is set four tiles in front of Pac-Man, if Pac-Man heads directly towards him, Pinky's target tile will actually be behind himself once they are less than four tiles apart. This will cause Pinky to choose to take any available turn-off in order to loop back around to his target. Because of this, it is a common strategy to momentarily "fake" back towards Pinky if he starts following closely. This will often send him off in an entirely different direction.

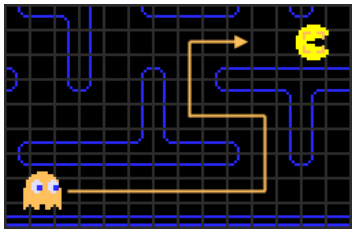
The Blue Ghost



The blue ghost is nicknamed Inky, and remains inside the ghost house for a short time on the first level, not joining the chase until Pac-Man has managed to consume at least 30 of the dots. His English personality description is bashful, while in Japanese he is referred to as 気紛れ, kimagure, or "whimsical". Inky is difficult to predict, because he is the only one of the ghosts that uses a factor other than Pac-Man's position and orientation when determining his target tile. Inky actually uses both Pac-Man's position and facing, as well as Blinky's (the red ghost's) position in his calculation. To locate Inky's target, we first start by selecting the position two tiles in front of Pac-Man in his current direction of travel, similar to Pinky's targeting method. From

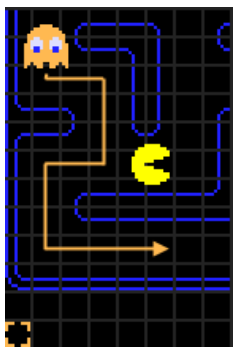
there, imagine drawing a vector from Blinky's position to this tile, and then doubling the length of the vector. The tile that this new, extended vector ends on will be Inky's actual target.

As a result, Inky's target can vary wildly when Blinky is not near Pac-Man, but if Blinky is in close pursuit, Inky generally will be as well. Note that Inky's "two tiles in front of Pac-Man" calculation suffers from exactly the same overflow error as Pinky's four-tile equivalent, so if Pac-Man is heading upwards, the endpoint of the initial vector from Blinky (before doubling) will actually be two tiles up and two tiles left of Pac-Man.

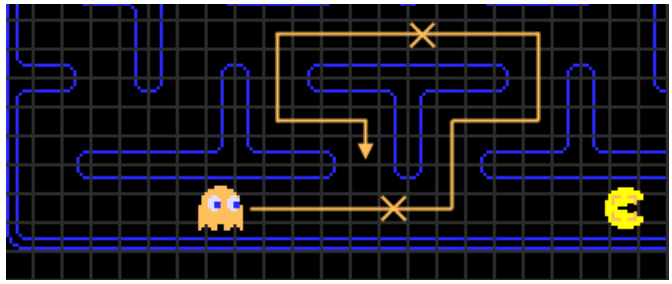


The Orange Ghost

The orange ghost, Clyde, is the last to leave the ghost house, and does not exit at all in the first level until over a third of the dots have been eaten. Clyde's English personality description is pokey, whereas the Japanese description is お惚け, otoboke or "feigning ignorance". As is typical, the Japanese version is more accurate, since Clyde's targeting method can give the impression that he is just "doing his own thing," without concerning himself with Pac-Man at all. The unique feature of Clyde's targeting is that it has two separate modes which he constantly switches back and forth between, based on his proximity to Pac-Man. Whenever Clyde needs to determine his target tile, he first calculates his distance from Pac-Man. If he is farther than eight tiles away, his targeting is identical to Blinky's, using Pac-Man's current tile as his target. But, as soon as his distance to Pac-Man becomes less than eight tiles, Clyde's target is set to the same tile as his fixed one in Scatter mode, just outside the bottom-left corner of the maze.



The combination of these two methods has the overall effect of Clyde alternating between coming directly towards Pac-Man, and then changing his mind and heading back to his corner whenever he gets too close. On the diagram



above, the X marks on the path represent the points where Clyde's mode switches. If Pac-Man somehow managed to remain stationary in that position, Clyde would indefinitely loop around that T-shaped area. As long as the player is not in the lower left corner of the maze, Clyde can be avoided completely by simply ensuring that you do not block his "escape route" back to his corner. While Pac-Man is within eight tiles of the lower left corner, Clyde's path will end up in exactly the same loop as he would eventually maintain in Scatter mode.

Wrapping Up

If you've made it this far, you should now have a fairly complete understanding of the logic behind Pac-Man's ghost movement. Understanding the ghosts's behavior is probably the single most important step towards becoming a skilled Pac-Man player, and even a general idea of where they are likely to move next should greatly improve your abilities. I've never been good at Pac-Man, but while I was researching this article and testing a few things, I found that I was able to avoid the ghosts much more easily than before. Even small things make a huge difference, such as recognizing a switch to Scatter mode and knowing that you have a few seconds where the ghosts won't (deliberately) try to kill you.

Pac-Man is an amazing example of seemingly complex behavior arising from only a few cleverly designed rules. The result being a deep and challenging game that players still strive to master, 30 years after its release. ■

Chad Birch is a gamer and programmer living in Calgary, Alberta, Canada. He distracts himself from his ever expanding lists of "games to play" and "programming projects to work on" by analyzing and writing about game mechanics at *gameinternals.com*.

Reprinted with permission of the original author. First appeared in *hn.my/pacman*.

WUFOO

Making web forms
easy + fast + fun

Will Freemium Work For You?

By MARTIN KLEPPMANN *and* RAHUL VOHRA

Ruben Gamez says freemium doesn't work [hn.my/free]. Yet Dropbox, MailChimp and Evernote say freemium does work [hn.my/freecase]. So, does freemium work or not?

This is the wrong question. The right question is: when does freemium work?

The business model for Rapportive (which adds rich contact profiles to your email) is simple: build an amazingly useful product, and charge money for it. It will most likely be freemium: a base version of Rapportive will always be free, and there will be a premium version which costs money. We think that this strategy will work for us.

But why? What signals indicate whether freemium will work? I think it depends on many things: the type of product you have, the kind of company you want to build, and more.

Ten Questions To Consider

❶ **How do users feel about your product as a function of time?**
Products which increase in value over time are good candidates for freemium.

Do users love your product for a short while, and then get bored of it? For example, casual games. Are they oblivious to it most of the time, but occasionally desperate for it? For example, data recovery tools. Do they always dislike it? For example, virus scanners. Or does it gradually get more useful over time?[1] For example, a tool which recovers corrupted photos from your camera is unlikely to work as freemium (cameras can corrupt files, but it happens rarely), whereas a tool such as Evernote works well as a freemium (it becomes more useful over time as you enter more data into it).

❷ **How good is your long-term retention?**
With good retention you have a better chance of converting free users to paid users.

Your product may become more useful over time, but if a large proportion of free users stop using it after a while that's bad news if you were hoping to convert them to paying later on. In that case, you should probably ask them for money early, while you still have their attention, or improve your retention! But, if a large proportion of your free users stick around for a long time, and your product becomes more useful over time, more and more of them will eventually convert from free to paid.

❸ Does your product require behavior change or can people start using it gradually?
If people can start using your product gradually, freemium might work.

For example, an image editor requires an abrupt change of behavior: you've got to stop using the old one and start using the new one. In that case, charging all users from day one may actually help you: by having spent money, users are more likely to also spend the time learning how to use the new application (due to the sunk cost fallacy), and thus end up getting more value from the product. On the other hand, if your product doesn't have a big learning overhead and users can gently start using it, it may be better to charge for it later.

❹ Does your product have distinct modes of use for different audiences?
If it does, you can be more creative with your freemium model.

I am fascinated by Yammer's business model. They get lots of people in an organization to use it for free, and then sometime later, when there are lots of active users, they sell the enterprise IT department on stuff like data ownership, admin controls, and security tools. The end-users who sign up do not care about the admin features — the IT department does, and they are a separate audience. This is an interesting take on freemium: always free for end users; always paid-for for IT departments doing their job.

❺ What is your market like?
If you are targeting a large unmet need, you should make your product free.

When faced with a wide open field you should be in land-grab mode, and acquire users before your competitors do. On the other hand, if you're in a well-established market, you'll need to gradually convince users to move away from their existing solutions. In this case, charging money is the best way of finding customers who care enough about the problem that they are willing to pay for a better solution.

❻ Are you targeting a premium niche?
Free users are the opposite of premium.

Sometimes free users are more troublesome than paid users (for example, MailChimp was faced with a spam problem when they started offering a free plan). If you are going for the top end of the market, giving something away for free may hurt you more than it benefits you. The price of your product says a lot about your positioning, and people tend to assume that if something is free, it's less valuable than something expensive.

❼ Which metric do you use to separate free from paying users?
Freemium makes sense if there's an obvious point to start charging.

Do your users need to pay when they exceed a certain number of seats, credits, or widgets? Picking a metric is tricky, and is a topic worthy of a separate article. Number of seats is a common metric, but it only makes sense for applications where there's a downside to everyone using the same log-in;for example, if your application is an analytics dashboard, it doesn't really matter if there's one shared log-in or each person has a separate log-in. If there is no obvious metric which separates free from paying users, you should probably charge everybody.

❽ Do you depend on word-of-mouth marketing?
More users (even if they are free) = more mouths.

If yes, note that more people using your product means more mouths to spread the word. You still need to reach the right kind of users, so the question is: can you reach people who will pay through the word of people who will remain forever free? My guess is that if you're targeting a specific niche, word-of-mouth spread and willingness to pay are strongly correlated (which suggests that there is little benefit in having lots of free users). If you're targeting a broad audience, the two are uncorrelated, so free users can help you carry the word to people who will pay.[2]

❾ What kind of company do you want to build?
You need lots of users if you want to take over the world.

If you've taken venture capital and want to take over the world, you need to grow quickly, even if it means leaving revenue on the table. If you want to grow organically and maximize profits, you're better off maximizing revenue per user, and ignoring those users who would never pay you anyway.

❿ What are your costs per user?
They had better be low if you have lots of free users.

Are your costs fixed (developers, testers) or variable (servers, support)? If your variable costs are low enough, it's fine to have a low conversion rate, because one paid-for user may pay for 1,000 free users. Support for free users is often the limiting factor. To keep your support burden low, you'll need to make your product easy to use and fix all your bugs…but that's well worth doing anyway!

So, why does Ruben say that free plans don't work? Well, for his situation, I think he's absolutely right. Let's consider the above questions for Bidsketch [bidsketch.com].

Bidsketch a workflow tool for designers: it's immediately valuable and then probably stays this valuable over time. It requires a change of behaviour: stop using email and start using Bidsketch. It has two distinct modes of use (one for designers and one for clients), but it seems pitched at one audience (designers). It competes with email, a well established solution. It seems to target a premium niche of the best freelance designers — this is not a tool for everybody. Ruben is self-declared Micropreneur who wants his business to grow organically. And, because he's working by himself (as far as I can tell), the support burden of free users would be significant.

So I agree: freemium is not right for Ruben's product. But it might be right for yours. ■

Notes

1. Phil Libin, CEO of Evernote, gave an excellent talk [hn.my/evernote] on Evernote's business metrics in May. It's packed with great insights for products which gradually become more useful over time:

> *"Every month, the longer you use [Evernote], the more valuable it gets. And since the long-term retention is flat, and the conversion goes up, what you see is: the longer a cohort stays, the more valuable they become." (at 14'55")*

Examining the cohort of users which signed up in March 2008, Phil found that after 3 months, they were making $300/month from 11,000 people (a conversion rate of about 0.6%); 22 months later, they were making $8,000/month from the same 11,000 people (a conversion rate of about 16%!):

> *"Users are kind of like a nice wine, or a stinky cheese. As [a cohort of users] ages, it actually gets better. A lot of the people who wouldn't pay leave, and a lot of the people who stay end up paying. Even though there's no hard sell, and you can use Evernote forever for free, a much larger percentage winds up converting." (at 17'05")*

Of course, not every application increases in value to users over time.

2. Rapportive has broad appeal and is not limited to a specific niche, so it makes sense for us to have lots of free users. When Brad Feld recently said that he was trying Gmail, a chorus of our users jumped to recommend Rapportive to him. We wouldn't have got that without lots of people using and loving our product.

Martin Kleppmann is co-founder of Rapportive. He likes to pretend that he can do a bit of everything, including architecting software, designing user experiences, building businesses, and writing geeky essays at *martin.kleppmann.com*.

Rahul Vohra is a co-founder and the CEO of Rapportive. He is a computer scientist, a gamer, and an entrepreneur. You can follow Rahul on Twitter at *@rahulvohra*.

#badass

On Not Hiring

By GABRIEL WEINBERG

HIRING IS HARD. Not hiring can seem even harder, but often isn't.

At my last company we went from entrance to exit without hiring one employee. I'm now three years into DuckDuckGo, and still haven't hired.

Needless to say, I'm an outlier. So, don't take what I say about hiring too seriously, but perhaps I have something useful to say on not hiring.

Most angel pitches I get seem to suggest the use of funds will go to salary — both to the founders and to immediate new hires. The assumption is, of course, that these new hires will move the company forward faster.

Yet every time I see that pitch, I look at my own experience and question this assumption. I would much rather see the use of initial funds around figuring out distribution, i.e., testing out different traction verticals. And then, once one-or-more customer acquisition channels are flowing, hire.

So why do people want to hire so early?

We need to build x, y, and z ASAP. Before you've figured out distribution? What evidence do you have that x, y, and z, once built, will make customer acquisition any easier?

Beta customers saying they want things isn't enough. There isn't a good reason to add x, y, and z to your product, i.e., complexity, unless you really know it will propel you faster to new customers. Yes, you can never really know, but I see a lot of people who certainly don't know.

But I understand their position. They like engineering, they like working on hard problems, and they like the idea of running a team. That's great, but it doesn't make it the right business decision.

We need a real designer because we suck at design. Have you really tried yet? Really? Usually not.

I'm not the world's best designer, by any means, nor am I "classically trained" in design, as they say, but I like my results, and I bring in the big guns when needed as freelancers. I'm sure you can do the same for at least your initial versions.

It just takes time and effort. Yes, that is time and effort you may not want to spend, but do you need to hire a full-time position because you're insecure or lazy? No.

Instead, lean on the powers of incremental improvement and the Pareto principle (80/20 rule). Spend time each week looking at specific parts of your design, and iterate on them. It will get better if you put in the time. And then for finishing touches, e.g., nicer images (the last 20%), outsource via 99designs or freelancers.

> **“You have a lot of short term needs, but that doesn’t mean they should turn into full-time positions.”**

A corollary to this one is user experience (UX), i.e., interaction design. I agree this is super important. I also still think the founders should be doing it. Again, iterate based on real feedback from users, and then bring in consultants and tools to give you ideas and polish.

We have too much to do. Any startup can easily grow to fill 100% of your time. That doesn’t mean you’re spending your time on the right things, or that hiring someone new and filling 100% of their time will increase outcome potential for your startup.

In addition, there are three main problems with hiring.

The wrong person can negatively impact your startup. There are horror stories, but more run of the mill is they’re just mediocre or don’t have a true startup mentality. Their presence can turn your company mediocre, and that is not good.

People also tend to underestimate the time it will require post-hiring to manage your hire(s). You’ve just added lots of meetings and other communication channels. Hiring takes a lot of time before and after. Your employee will not be inside your head.

And finally, hiring takes money. It increases your burn rate significantly. Companies before product fit, i.e., traction, need to stay around long enough until they get it. That can take a lot of time — like years. There are countless cases where companies folded only to miss their moment and see other companies rise up where they might have done so.

One approach I like that some of my portfolio companies are taking is to tie hiring decision points to traction milestones, i.e., once we hit $xK/month in revenue we’ll do our next hire.

The nice things about this approach are that it allows you to (a) manage the burn rate issue, and (b) take a long time to plan your hire. The latter allows you to make sure you’re getting the right person in the right position, and that they will have a positive impact on the startup.

Early on it is not entirely clear who that right position will turn out to be. You have a lot of short term needs, but that doesn’t mean they should turn into full-time positions. ■

Gabriel Weinberg is the founder of Duck Duck Go, a search engine. He is also an active angel investor, based out of Valley Forge, PA. More info at his homepage: *ye.gg*.

Reprinted with permission of the original author. First appeared in *hn.my/nothire*.

How To Become An Open-Source Contractor

By EVAN MILLER

I GET EMAILS FROM people I've never met asking if they can pay me money to improve some of my favorite open-source software. That's right. I get money. I get to do something I enjoy. I get good software. And I get to share it with whoever I please. It's one of the greatest economic arrangements in the world. It's up there with being landed gentry, a corrupt government official, or Dave Barry.

This essay is about how you, too, can get paid to write great open source software. If you're not a software developer, you should probably spend the next few minutes of your leisure time on things other than this article. I recommend The Really Big Button That Doesn't Do Anything.

Now, when I talk about being paid to hack on open source projects, I'm not talking about sucking the big scaly teat of the Mozilla Foundation. That would be like having any other job, except there's no Sally Shareholder holding Mindy Management's tookus over the teakettle. Nor am I talking about hiding out in the "community" wing of the Heavily Underfunded Projects Division of, say, Sun Microsystems or Apple Computers. I mean waking up, going to the computer, and banging out some code before breakfast. I mean turning down projects that don't sound like fun. I mean getting paid in proportion to output. By another name: contract work.

I didn't really set out with the goal of being an open source contractor. It just sort of happened, like sex with a much older neighbor. If I wanted to be a good contractor, a full-time contractor, there's probably a lot more I could do. I'll get to that. What follows is a mix of tested and untested advice. I think it's all pretty good. The road ain't easy and it ain't for everyone.

First, you need to understand some economics. It's hard to make money from open source software. Some companies have tried giving away software and then selling "support and services", but that business plan has proven as profitable as giving away VCRs and selling the Owner's Manuals.

If you write proprietary software, you can sell thousands of copies. If you write open source software, you can sell the first copy, but you're forced to give away all subsequent copies. So, as an open source contractor, you will not be writing software for Connie Consumer. No Pidgin plugins, no Gnome widgets, no iPod synchronizer, or whatever. You will be writing software for Seymour CTO: a single company that stands to save serious money by your efforts.

> **“If you want to be a successful contractor, you have to make yourself a scarce resource.”**

It’s a peculiar set of circumstances that will allow you to write open source software for a company. The company needs to have a culture of fixing problems themselves. This usually means fixing problems at any hour, day or night. If software bugs can “wait till next week,” then a company would probably be better off buying software from a vendor than developing software in-house or using an open source solution. In general, proprietary software can be developed with greater resources.

The need for quick fixes I think explains why open-source software has thrived on web servers. Always-on web companies don’t have to rely on another company to fix their problems. Many Internet companies bring in thousands of dollars an hour, and every minute the site is broken means Stanley Stakeholder’s net worth inches downward. An Internet company simply can’t afford to wait for another company to wake up and diagnose the problem.

A company reliant on open-source software for revenue will be large enough to have developers on hand (at the very least to fix problems), but not so large that they want to develop everything themselves. The company that will hire an open source contractor has at least one but probably no more than a few dozen software engineers.

But hiring you, the contractor, is a risk. You might not deliver on time, your software might crash, and you might flee the country. In general, a company would much prefer an employee did the work than a contractor, because an employee has more to lose if he does a bad job. So, why would a firm with software engineers on the payroll hire another software engineer on a contract basis? *Because that’s all they can afford*.

If you want to be a successful contractor, you have to make yourself a scarce resource. You need to be able to do something better than almost all of the software engineers at a particular firm. You want engineering managers to think, “It’d be great if we could hire this guy, but at the very least let’s try to get some of his time.”

The trick, of course, is to become an expert on something. Preferably a world expert. This doesn’t mean you have to know something better than anyone in the world; instead, it just means you can demonstrate your expertise as well as anyone else in the same domain. You don’t need to prove you’re the best. You just want no-one else to have proven that they’re the best.

I became an expert by accident. Gather around, beardless youth, I shall tell you a tale. Once, long ago (2006), I worked for A Large Internet Retailer. My coworkers

> **If you're a developer who has never done contract work, and never been paid expert wages, you are in for a ride.**

often talked about things I didn't understand — "non-blocking sockets" and "duping file descriptors," and other concepts that sounded vaguely like fighting moves. In an effort to decode this strange tongue, I did some Google searches and found out my colleagues were discussing, not Marquess of Queensbury rules, or Masonic arcana, but the closely related subject of UNIX network programming. I set out to learn more, primarily so I could fit in. I read a book on UNIX and a book on networking, and tinkered on my machine at home. But I found it hard to write even rudimentary server programs. They would work, mostly, but contain bugs when talking with certain web browsers. So, I decided I'd let someone else's program deal with the client bugs.

At the time, I happened to be playing with Nginx — a web server with a funny name. I decided I would figure out how to write my toy programs as extensions to Nginx, rather than as standalone programs. Unfortunately, I couldn't find any documents on the web about how to write Nginx extensions, so I poked around the code and jotted down observations in a notebook. After a several evenings and couple Saturdays, I had a pretty clear idea of how to write an extension. Figuring out the details was much harder than I initially expected; the Nginx source code, being of Russian origin, had very few comments in English, and very many variables with one-character or two-character names, or slightly longer but equally obscure variable names such as "nldcf".

My first Nginx extension, culminating from weeks of effort, was around 100 lines of code. Most of the effort went into my notes, of which I had accumulated several pages. Because I didn't have anything better to do one afternoon, I typed up my notes and thought, "What the hell, I'll put them online." Mind you at this time I didn't have a homepage, so I simply sent to the Nginx mailing list a link to "Emiller's Balls Out Guide to Nginx Module Development (DRAFT)."

My guide was an instant hit in Russia — I was tickled to see the phrase "Emiller's Balls Out Guide" surrounded by Cyrillic text on several Russian tech sites. And in the English-speaking world, I started fielding a trickle of questions about Nginx internals. Apparently no-one this side of Saint Petersburg had written an Nginx module before, and all of a sudden I was an expert in this limited domain.

And apparently people needed an expert. I got an email a few days after publishing my guide asking if I wanted to write an Nginx module for an Internet startup. They had plenty of developers, the CTO explained, but rather than have one of them spend two weeks learning how to write an Nginx module using my guide, it would be more economical to pay me — you know, an expert — to write it. And they wanted the resulting work to be open source. At the time I really wanted a MacBook Pro, so I named a price equal to a MacBook Pro plus income tax, and they agreed.

If you're a developer who has never done contract work, and never been paid expert wages, you are in for a ride. I probably worked harder on that first paid module than anything I've worked on in my life. It felt liberating to know the faster I worked, the sooner I got paid. It felt good, yes, but greedy. Liberating isn't quite the right word. It felt motivating, energizing, inebriating; it hit the same spot as a double shot

> **"In the chef's kitchen, the knife that cuts anything, cuts nothing."**

of love and coke. It was like my brain opened up a brand new Reward Center twice as big as the old one — with triple the parking. It felt good. Good as gold. Good as greed. Good as God. I was Hernán Cortés, and C was my sword.

I delivered the final product in four days instead of four weeks, and had to spend a couple of days in Code Detox (i.e., watching lots of House). When I went to pick up my check, the CTO said, the company liked what I did so much they were giving me an extra 500 bucks just to say thanks.

I was getting busier with other projects at the time, and I declined their offer to contract more work. Since that first job, I've taken on gigs here and there — just enough to keep my skills sharp — but as they say, there's nothing like your first hit.

Anyway, enough of this ancient history. I'll try to distill my experience into a finite list of — O wisdom, what shall be thy verbal chains? — original aphorisms with commentary.

In the chef's kitchen, the knife that cuts anything, cuts nothing. To be a contractor, you have to specialize. "Smart and gets things done" ain't enough. You need to do something better than most people at the company that's hiring you. Pick a system, program, or set of libraries to specialize in. Pick something that interests you. But also pick something which small to medium-sized companies rely on. Write patches, features, applications, and plug-ins until you reckon you're as good at it as anybody out there. Then prove it.

A good teacher is better than a great student. To become known as an "expert", write about what you know about. Explain things in tutorials. Participate in mailing lists. Present at conferences. But remember...

Bigger locks keep smaller secrets. Don't give away all your knowledge. It's OK to keep some voodoo up your sleeve. It's even better to mention offhand that you're leaving out some "details".

More seek a charlatan at his office than a doctor at his home. Make a presence for yourself. Make a website explaining what you do. People aren't going to find you by telepathy.

Today a stitch, tomorrow a suit. Companies hiring you might have a few projects in mind, and they'll start with the smallest one just to test the waters. If it's a small job, take any price you can get. If you impress them, you'll be offered a much more lucrative job next.

And, finally, some non-aphoristic advice:

Look professional. Give out business cards at conferences. Make an invoice template. State your rates with confidence. Deliver ahead of schedule. Write tests and documentation. Code exactly what your client wants, and not what you think the world wants. Spell properly.

The world has room for open-source independent contractors. I don't know how much room. And I don't know for how long. But now you know how I did it.

Well, except for one or two details...■

Evan Miller is a graduate student in Economics at the University of Chicago, and the author of the Chicago Boss web framework.

Reprinted with permission of the original author. First appeared in *hn.my/contractor.*

How To Write A Simple Operating System

By MIKE SAUNDERS

THIS DOCUMENT SHOWS you how to write and build your first operating system in x86 assembly language. It explains what you need, the fundamentals of the PC boot process and assembly language, and how to take it further. The resulting OS will be very small (fitting into a boot loader) and have very few features, but it's a starting point for you to explore further.

After you have read the guide, see the MikeOS project [mikeos.berlios.de] for a bigger x86 assembly language OS that you can explore to expand your skills.

Requirements

Prior programming experience is essential. If you've done some coding in a high-level language like PHP or Java, that's good, but ideally you'll have some knowledge of a lower-level language like C, especially on the subject of memory and pointers.

For this guide we're using Linux. OS development is certainly possible on Windows, but it's so much easier on Linux as you can get a complete development toolchain in a few mouse clicks/commands. Linux is also very good for making floppy disk and CD-ROM images — you don't need to install loads of fiddly programs.

Installing Linux is very easy these days: grab Ubuntu and install it in VMware or VirtualBox if you don't want to dual-boot. When you're in Ubuntu, get all the tools you need to follow this guide by entering this in a terminal window:

```
sudo apt-get install build-essential qemu nasm
```

This gets you the development toolchain (compiler, etc), QEMU PC emulator and the NASM assembler, which converts assembly language into raw machine code executable files.

PC primer

If you're writing an OS for x86 PCs (the best choice, due to the huge amount of documentation available), you'll need to understand the basics of how a PC starts up. Fortunately, you don't need to dwell on complicated subjects such as graphics drivers and network protocols, as you'll be focusing on the essential parts first.

When a PC is powers up, it starts executing the **BIOS** (Basic Input/Output System), which is essentially a mini-OS built into the system. It performs a few hardware tests (e.g., memory checks), and typically spurts out a graphic (e.g., Dell logo) or diagnostic text to the screen. Then, when it's done, it starts to load your operating system from any media it can find. Many PCs jump to the hard drive and start executing code they find in the **Master Boot Record** (MBR), a 512-byte section at the start of the hard drive; some try to find executable code on a floppy disk (boot sector) or CD-ROM.

This all depends on the boot order — you can normally specify it in the BIOS options. The BIOS loads 512 bytes from the chosen media into its memory, and begins executing it. This is the boot loader, the small program that then loads the main OS kernel or a larger boot program (e.g., GRUB/LILO for Linux systems). This 512-byte boot loader has two special numbers at the end to tell the OS that it's a boot sector - we'll cover that later.

Note that PCs have an interesting feature for booting. Historically, most PCs had a floppy drive, so the BIOS was configured to boot from that device. But, today many PCs don't have a floppy drive — only a CD-ROM — so a hack was developed to cater for this. When you're booting from a CD-ROM, it can **emulate a floppy disk** — the BIOS reads the CD-ROM drive, loads in a chunk of data, and executes it as if it was a floppy disk. This is incredibly useful for us OS developers, as we can make floppy disk versions of our OS, but still boot it on CD-only machines. (Floppy disks are really easy to work with, whereas CD-ROM filesystems are much more complicated.)

So, to recap, the boot process is:

1. Power on: the PC starts up and begins executing the BIOS code.
2. The BIOS looks for various media such as a floppy disk or a hard drive.
3. The BIOS loads a 512-byte boot sector from the specified media and begins executing it.
4. Those 512 bytes then go on to load the OS itself, or a more complex boot loader.

For MikeOS, we have the 512-byte boot loader, which we write to a floppy disk image file (a virtual floppy). We can then inject that floppy image into a CD for PCs that only have CD-ROM drives. Either way, the BIOS loads it as if it was on a floppy, and starts executing it. We have control of the system!

Assembly language primer

Most modern operating systems are written in C/C++. That's very useful when portability and code-maintainability are crucial, but it adds an extra layer of complexity to the proceedings. For your very first OS, you're better off sticking with assembly language, as used in MikeOS. It's more verbose and non-portable, but you don't have to worry about compilers and linkers. Besides, you need a bit of assembly to kick-start any OS.

Assembly language (or colloquially "asm") is a textual way of representing the instructions that a CPU executes. For instance, an instruction to move some memory in the CPU may be `11001001 01101110`, but that's hardly memorable! So, assembly provides mnemonics to substitute for these instructions, such as `mov ax, 30`. They correlate directly with machine-code CPU instructions, but without the meaningless binary numbers.

Like most programming languages, assembly is a list of instructions followed in order. You can jump around between various places and set up subroutines/functions, but it's much more minimal than C# and friends. You can't just print "Hello world" to the screen — the CPU has no concept of what a screen is! Instead, you work with memory: manipulating chunks of RAM, performing arithmetic on them, and putting the results in the right place. Sounds scary? It's a bit alien at first, but it's not hard to grasp.

At assembly-level, there is no such thing as variables in the high-level language sense. What you do have, instead, is a set of **registers**, which are on-CPU memory stores. You can put numbers into these registers and perform calculations on them. In 16-bit mode, these registers can hold numbers between 0 and 65535. Here's a list of the fundamental registers on a typical x86 CPU:

AX, BX, CX, DX	General-purpose registers for storing numbers that you're using. For instance, you may use AX to store the character that has been pressed on the keyboard, while using CX to act as a counter in a loop. (Note: these 16-bit registers can be split into 8-bit registers such as AH/AL, BH/BL, etc.)
SI, DI	Source and destination data index registers. These point to places in memory for retrieving and storing data.
SP	The Stack Pointer (explained in a moment).
IP (sometimes CP)	The Instruction/Code Pointer. This contains the location in memory of the instruction being executed. When an instruction has finished, it is incremented and moves on to the next instruction. You can change the contents of this register to move around in your code.

So, you can use these registers to store numbers as you work — a bit like variables, but they're much more fixed in size and purpose. There are a few others, notably **segment registers**. Due to limitations in old PCs, memory was handled in 64k chunks called segments. This is a really messy subject, but thankfully you don't have to worry about it. For the time being, your OS will be less than a kilobyte anyway! In MikeOS, we limit ourselves to a single 64k segment so that we don't have to mess around with segment registers.

The **stack** is an area of your main RAM used for storing temporary information. It's called a stack because numbers are stacked on top of one another. Imagine a Pringles tube: if you put in a playing card, an iPod Shuffle, and a beermat, you'll pull them out in the reverse order (the beermat, then the iPod, and, finally, the playing card). It's the same with numbers: if you push the numbers 5, 7, and 15 onto the stack, you will pop them out as 15 first, then 7, and then 5. In assembly, you can **push** registers onto the stack and **pop** them out later — it's useful when you want to store the value of a register temporarily while you use that register for something else.

PC **memory** can be viewed as a linear line of pigeonholes ranging from byte 0 to whatever you have installed (millions of bytes on modern machines). At byte number 53,634,246 in your RAM, for instance, you may have your web browser code to view this document. But whereas we humans count in powers of 10 (10, 100, 1000, etc — in decimal), computers are better off with powers of two (because they're based on binary). So we use **hexadecimal**, which is **base 16**, as a way of representing numbers. See this chart to understand:

Decimal	0	1	2	3	4	5	6	7	8	9	
Hexadecimal	0	1	2	3	4	5	6	7	8	9	
Decimal	10	11	12	13	14	15	16	17	18	19	20
Hexadecimal	A	B	C	D	E	F	10	11	12	13	14

As you can see, whereas our normal decimal system uses 0-9, hexadecimal uses 0-F in counting. It's a bit weird at first, but you'll get the hang of it. In assembly programming, we identify hexadecimal (hex) numbers by tagging an "**h**" onto the end — so **0Ah** is hex for the number **10**. (You can also denote hexadecimal in assembly by prefixing the number with 0x - for instance, 0x0A.)

Let's finish off with a few common assembly instructions. These move memory around, compare them, and perform calculations. They're the building blocks of your OS — there are hundreds of instructions, but you don't have to memorize them all, because the most important handful are used 90% of the time.

mov	Copies memory from one location (or register) to another. For instance, `mov ax, 30` places the number 30 into the AX register. Using square brackets, you can get the number at the memory location pointed to by the register. For instance, if BX contains 80, then `mov ax, [bx]` means "get the number in memory location 80, and put it into AX". You can move numbers between registers too: `mov bx, cx`.
add/sub	Adds a number to a register. `add ax, FFh` adds FF in hexadecimal (255 in our normal decimal) to the AX register. You can use `sub` in the same way: `sub dx, 50`.
cmp	Compares a register with a number. `cmp cx, 12` compares the CX register with the number 12. It then updates a special register on the CPU called **FLAGS** — a special register that contains information about the last operation. In this case, if the number 12 is bigger than the value in CX, it generates a negative result, and notes that negative in the FLAGS register. We can use this in the following instructions.
jmp/jg/jl...	Jump to a different part of the code. `jmp label` jumps (GOTOs) to the part of our source code where we have label: written. But, there's more — you can jump conditionally, based on the CPU flags set in the previous command. For instance, if a `cmp` instruction determined that a register held a smaller value than the one with which it was compared, you can act on that with `jl label` (jump to label if less than). Similarly, `jge label` jumps to "label" in the code if the value in the `cmp` was greater than or equal to its compared number.
int	Interrupt the program and jump to a specified place in memory. Operating systems set up **interrupts** that are analogous to subroutines in high-level languages. For instance, in MS-DOS the 21h interrupt provides DOS services (e.g., as opening a file). Typically, you put a value in the AX register, then call an interrupt and wait for a result (passed back in a register too). When you're writing an OS from scratch, you can call the BIOS with `int 10h`, `int 13h`, `int 14h`, or `int 16h` to perform tasks like printing strings, reading sectors from a floppy disk, etc.

Let's look at some of these instructions in a little more detail. Consider the following code snippet:

```
        mov bx, 1000h
        mov ax, [bx]
        cmp ax, 50
        jge label
        ...
label:
        mov ax, 10
```

In the first instruction, we move the number 1000h into the BX register. Then, in the second instruction, we store in AX whatever is in the memory location pointed to by BX. This is what the `[bx]` means: if we just did `mov ax, bx` it'd simply copy the number 1000h into the AX register. But, by using square brackets, we're saying: don't just copy the contents of BX into AX, but copy the contents of the memory address to which BX points. Given that BX contains 1000h, this instruction says: find whatever is at memory location 1000h, and put it into AX.

So, if the byte of memory at location 1000h contains 37, then that number 37 will be put into the AX register via our second instruction. Next up, we use `cmp` to compare the number in AX with the number 50 (the decimal number 50 — we didn't suffix it with "h"). The following `jge` instruction acts on the `cmp` comparison, which has set the FLAGS register as described earlier. The `jge label` says: if the result from the previous comparison is greater than or equal, jump to the part of the code denoted by `label:`. So, if the number in AX is greater than or equal to 50, execution jumps to `label:`. If not, execution continues at the "..." stage.

One last thing, you can insert data into a program with the `db` (define byte) directive. For instance, this defines a series of bytes with the number zero at the end, representing a string:

```
mylabel: db 'Message here', 0
```

In our assembly code, we know that a string of characters, terminated by a zero, can be found at the `mylabel:` position. We could also set up single byte to use somewhat like a variable:

```
foo: db 0
```

Now `foo:` points at a single byte in the code, which in the case of MikeOS will be writable as the OS is copied completely to RAM. So you could have this instruction:

```
mov byte al, [foo]
```

This moves the byte pointed to by `foo` into the AL register.

That's the essentials of x86 PC assembly language, and enough to get you started. When writing an OS, though, you'll need to learn much more as you progress, so see the Resources section at *mikeos.berlios.de* for links to more in-depth assembly tutorials.

Your first OS

Now you're ready to write your first operating system kernel! Of course, this is going to be extremely bare bones, just a 512-byte boot sector as described earlier, but it's a starting point for you to expand further. Paste the following code into a file called **myfirst.asm** and save it into your home directory — this is the source code to your first OS:

```
  BITS 16
start:
  mov ax, 07C0h          ; Set up 4K stack space after this boot loader
  add ax, 288            ; (4096 + 512) / 16 bytes per paragraph
  mov ss, ax
  mov sp, 4096

  mov ax, 07C0h          ; Set data segment to where we're loaded
  mov ds, ax

  mov si, text_string    ; Put string position into SI
  call print_string      ; Call our string-printing routine

  jmp $                  ; Jump here - infinite loop!

  text_string db 'This is my cool new OS!', 0

print_string:            ; Routine: output string in SI to screen
  mov ah, 0Eh            ; int 10h 'print char' function
.repeat:
  lodsb                  ; Get character from string
  cmp al, 0
  je .done               ; If char is zero, end of string
  int 10h                ; Otherwise, print it
  jmp .repeat
.done:
  ret

  times 510-($-$$) db 0 ; Pad remainder of boot sector with 0s
  dw 0xAA55              ; The standard PC boot signature
```

Let's step through this. The `BITS 16` line isn't an x86 CPU instruction — it just tells the NASM assembler that we're working in 16-bit mode. NASM can then translate the following instructions into raw x86 binary. Then we have the `start:` label, which isn't strictly needed as execution begins right at the start of the file anyway, but it's a good marker.

From here onwards, note that the semicolon (;) character is used to denote non-executable text comments — we can put anything there.

The following six lines of code aren't really of interest to us — they simply set up the segment registers, so that the stack pointer (SP) knows where our handy stack of temporary data is, and where the data segment (DS) is located. As mentioned, segments are a hideously messy way of handling memory from the old 16-bit days, but we just set up the segment registers and forget about them. (The references to 07C0h are the equivalent segment location at which the BIOS loads our code, so we start from there.)

The next part is where the fun happens. The `mov si, text_string` line says, "Copy the location of the text string below into the SI register." Simple enough! Then we use `call`, which is like a GOSUB in BASIC or a function call in C. It means, jump to the specified section of code, but prepare to come back here when we're done.

How does the code know how to do that? Well, when we use a `call` instruction, the CPU increments the position of the IP (Instruction Pointer) register and **pushes** it onto the stack. You may recall from the previous explanation of the stack that it's a last-in first-out memory storage mechanism. All that business with the stack pointer (SP) and stack segment (SS) at the start cleared a space for the stack, so that we can drop temporary numbers there without overwriting our code.

So, the `call print_string` means, jump to the print_string routine, but push the location of the next instruction onto the stack, so we can `pop` it off later and resume execution here. Execution has jumped over to `print_string:`, this routine uses the BIOS to output text to the screen. First we put 0Eh into the AH register (the upper byte of AX). Then we have a `lodsb` (load string byte) instruction, which retrieves a byte of data from the location pointed to by SI, and stores it in AL (the lower byte of AX). Next we use `cmp` to check if that byte is zero — if so, it's the end of the string and we quit printing (jump to the `.done` label).

If it's not zero, we call `int 10h` (interrupt our code and go to the BIOS), which reads the value in the AH register (0Eh) we set up before. Ah, says the BIOS. 0Eh in the AH register says , "Print the character in the AL register to the screen!" So, the BIOS prints the first character in our string, and returns from the int call. We then jump to the `.repeat` label, which starts the process again. `lodsb` loads the next byte from SI (it increments SI each time), see if it's zero and decide what to do. The `ret` at the end of our string-printing routine says, "We've finished here — return back to the place where we were `call`ed by popping the code location from the stack back into the IP register".

So, there we have a demonstration of a loop in a stand-alone routine. You can see that the `text_string` label is alongside a stream of characters, which we insert into our OS using `db`. The text is in apostrophes so that NASM knows it's not code, and at the end we have a zero to tell our `print_string` routine that we're at the end.

Let's recap: we start off by setting up the segment registers so that our OS knows where the stack pointer and executable code resides. Then, we point the SI register at a string in our OS binary, and `call` our string-printing routine. This routine scans through the characters pointed to by SI and displays them until it finds a zero, at which point it `ret`urns back into the code that `call`ed it. Then, the `jmp $` line means, keep jumping to the same line. (The "$" in NASM denotes the current point of code.) This sets up an infinite loop, so that the message is displayed and our OS doesn't try to execute the following string!

The final two lines are interesting. For a PC to recognize a valid floppy disk boot sector, it has to be exactly 512 bytes in size and end with the numbers AAh and 55h (the boot signature). So, the first of these lines says, "Pad out our resulting binary file to be 510 bytes in size." Then the second line uses `dw` (define a word — two bytes) containing the aforementioned boot signature. Voilà! A 512-byte boot file with the correct numbers at the end for the BIOS to recognize.

Let's build our new OS. In a terminal window, in your home directory, enter:

```
nasm -f bin -o myfirst.bin myfirst.asm
```

Here we assemble the code from our text file into a raw binary file of machine code instructions. With the `-f bin` flag, we tell NASM that we want a plain binary file (not a complicated Linux executable — we want it as plain as possible!) The `-o myfirst.bin` part tells NASM to generate the resulting binary in a file called myfirst.bin.

Now we need a virtual floppy disk image to which we can write our boot loader-sized kernel. Copy **mikeos.flp** from the **disk_images/directory** of the MikeOS bundle into your home directory, and rename it **myfirst.flp**. Then, enter:

```
dd status=noxfer conv=notrunc if=myfirst.bin
of=myfirst.flp
```

This uses the "dd" utility to directly copy our kernel to the first sector of the floppy disk image. When it's done, we can boot our new OS using the QEMU PC emulator as follows:

```
qemu -fda myfirst.flp
```

And there you are! Your OS will boot up in a virtual PC. If you want to use it on a real PC, you can write the floppy disk image to a real floppy and boot from it, or generate a CD-ROM ISO image. For the latter, make a new directory called **cdiso** and move the **myfirst.flp** file into it. Then, in your home directory, enter:

```
mkisofs -o myfirst.iso -b myfirst.flp cdiso/
```

This generates a CD-ROM ISO image called **myfirst.iso** that emulates a floppy disk and is bootable, using the virtual floppy disk image from before. Now you can burn that ISO to a CD-R and boot your PC from it! (Note that you need to burn it as a direct ISO image and not just copy it onto a disc.)

Next you'll want to improve your OS — explore the MikeOS source code to get some inspiration. Remember that boot loaders are limited to 512 bytes, so if you want to do a lot more, you'll need to make your boot loader load a separate file from the disk and begin executing it, in the same fashion as MikeOS. ■

Mike Saunders is a Linux journalist, machine code hacker and expert at Mario Kart (the SNES version). His operating system, MikeOS, can be found at *mikeos.berlios.de*. He lives in Bath, England.

Reprinted with permission of the original author. First appeared in *hn.my/os*.

Commentary

By MICHAEL MELANSON (a-priori)

WHILE IT'S COOL and all to write your own boot loader, this means you have to deal with a lot of the really ugly bits of x86/PC architecture like the A20 gate (if you don't know what that is, count yourself lucky.)

Instead, I suggest using GRUB to boot your kernel image. It leaves you in 32-bit mode and a relatively sane state. It's not hard to write a loader file (in assembly) which contains the multi-boot header and an entry point. Presumably, you'll want to set up a basic C runtime environment and call your C "main" function.

By JACQUES MATTHEIJ (jasquesm)

WRITING YOUR OWN boot loader is actually an excellent exercise in getting to know the intricacies of the memory mapping on the PC architecture. After you're done you can always decide to go for a readymade one, but rolling your own is definitely useful if you plan on writing your own OS.

NIH applies like always, but if you plan on taking control of the machine you might as well begin at the beginning. Rolling your own BIOS would be a step too far I think. :)

I Can Crack Your App With Just A Shell

(And How To Stop Me)

By KENNETH BALLENEGGER

Well, not you specifically, but by 'you' I mean the average Mac developer. It's too easy to crack Mac applications. Way too easy. By walking through how I can hack your application with only one Tterminal shell, I hope to shed some light on how this is most commonly done, and hopefully convince you to protect yourself against me. I'll be ending this article with some tips to prevent this kind of hack.

Disclaimer: I am fervently *against* software piracy, and I do not participate in piracy. Some will view this article as an endorsement of piracy, but rest assured that it is not. ButHowever, I do not believe that obscurity and ignoring the problem is an acceptable solution.

In order to follow along you're going to need a few command line utilities. You're going to need the Xcode tools installed,. andAnd lastly, you're going to need an application to operate on. I chose Exces [excesapp.com], a shareware aApplication I wrote a long time ago.

Let's start by making sure we have the two utilities we need: `otx` and `class-dump`. I like to use `Homebrew` as my package manager of choice. Note that I will use command-line utilities only, including `vim`. If you prefer GUIs, feel free to use your code-editor of choice, HexFiend, and `otx`'s GUI app.

```
$ sudo brew install otx
$ sudo brew install class-dump
```

The first step is to poke into the target application's headers, gentlemanly left intact by the unwitting developer.

```
$ cd Exces.app/Contents/MacOS
$ class-dump Exces | vim
```

Browse around, and find the following gem:

```
@interface SSExcesAppController : NSObject
{
[...]
        BOOL registred;
[...]
- (void)verifyLicenseFile:(id)arg1;
- (id)verifyPath:(id)arg1;
- (BOOL)registred;
```

What do we have here?! A (badly spelled) variable and what looks like three methods related to registration. We can now focus our efforts around these symbols. Let's continue poking by disassembling the source code for these methods.

```
$ otx Exces -arch i386
```

Note that Exces is a universal binary, and that we need to ensure we only deal with the active architecture. In this case, Intel's i386. Let us find out what `verifyLicenseFile:` does.

```
-(void)[SSExcesAppController verifyLicenseFile:]
[...]
+34  0000521e  e8c21e0100    calll 0x000170e5
-[(%esp,1) verifyPath:]
+39  00005223  85c0          testl %eax,%eax
+41  00005225  0f84e2000000  je    0x0000530d
[...]
+226 000052de  c6472c01      movb  $0x01,0x2c(%edi)
(BOOL)registred
[...]
```

This is not straight Objective-C code, but rather assembly — what C compiles into. The first part of each line, the offset, `+34`, shows how many bytes into the method the instruction is. `0000521e` is the address of the instruction within the program. `e8c21e0100` is the instruction in byte code. `calll 0x000170e5` is the instruction in assembly language. `-[(%esp,1) verifyPath:]` is what `otx` could gather the instruction to represent in Objective-C from the symbols left within the binary.

With this in mind, we can realize that `verifyLicenseFile:` calls the `methodverifyPath:` and later sets the boolean instance variable as `registred`. We can guess that `verifyPath:` is probably the method that checks the validity of a license file. We can see from the header that `verifyPath:` returns an object and thus would be way too complex to patch. We need something that deals in booleans.

Let's launch Exces in the `gdb` debugger and check when `verifyLicenseFile:` is called.

```
$ gdb Exces
(gdb) break [SSExcesAppController verifyLicenseFile:]
Breakpoint 1 at 0x5205
(gdb) run
```

No bite. The breakpoint is not hit on startup. We can assume that there's a good reason `whyverifyLicenseFile:` and `verifyPath:` are two separate methods. While we could patch `verifyLicenseFile:` to always set `registred` to true, `verifyLicenseFile:` is probably called only to check license files entered by the user. Quit `gdb` and let's instead search for another piece of code that calls `verifyPath:`. In the `otx` dump, find the following `inawakeFromNib:`,

```
-(void)[SSExcesAppController awakeFromNib]
[...]
+885  00004c8c  a1a0410100  movl   0x000141a0,%eax
verifyPath:
+890  00004c91  89442404    movl   %eax,0x04(%esp)
+894  00004c95  e84b240100  calll  0x000170e5
-[(%esp,1) verifyPath:]
899  00004c9a  85c0        testl  %eax,%eax
+901  00004c9c  7409        je     0x00004ca7
+903  00004c9e  8b4508      movl   0x08(%ebp),%eax
+906  00004ca1  c6402c01    movb   $0x01,0x2c(%eax)
(BOOL)registred
+910  00004ca5  eb7d        jmp    0xe00004d24
return;
[...]
```

The code is almost identical to `verifyLicenseFile:`. Here's what happens,

- `verifyPath:` is called. (`+894 calll`)
- A test happens based on the result of the call. (`+899 testl`)
- Based on the result of the text, jump if equal. (`+901 je`) A `test` followed by a `jeje` or `jnejne` (jump if not equal) is assembly-speak for an if statement.
- The `registred` ivar is set, if we have not jumped away.

Since `awakeFromNib` is executed at launch, we can safely assume that if we override this check, we can fool the application into thinking it's registered. The easiest way to do that is to change the `je` into a `jne`, essentially reversing its meaning.

Search the dump for any `jne` statement, and compare it to the `je:`,

```
+901  00004c9c  7409  je          0x00004ca7
+14   00004d9f  7534  jne         0x00004dd5
return;
```

`7409` is the binary code for `je 0x00004ca7`. `7534` is a similar binary code. If we simply switch the binary code for the `je` to `7534`, at address `00004c9c`, we should have our crack. Let's test it out in `gdb`.

```
$ gdb Exces
(gdb) break [SSExcesAppController awakeFromNib]
Breakpoint 1 at 0x4920
(gdb) r
(gdb) x/x 0x00004c9c
0x4c9c <-[SSExcesAppController awakeFromNib]+901>:
0x458b0974
```

We break on `awakeFromNib` so we're able to fiddle around while the application is frozen. `x/x` reads the code in memory at the given address. Now here's the confusing thing to be aware of: *endianness*. While on disk, the binary code is normal, Intel is a little-endian system which puts the most significant byte last, and thus reverses every four-byte block in memory. So, while the code at address `0x4c9c` is printed as `0x458b0974`, it's actually `0x74098b45`. We recognize the first two bytes `7409` from earlier.

We need to switch the first two bytes to `7534`. Let's start by disassembling the method so we can better see our way around. Find the relevant statement,

```
0x00004c9c <-[SSExcesAppController awakeFromNib]+901>:
je  0x4ca7 <-[SSExcesAppController awakeFromNib]+912>
```

Now let's edit code in memory,

```
(gdb) set {char}0x00004c9c=0x75
(gdb) x/x 0x00004c9c
0x4c9c <-[SSExcesAppController awakeFromNib]+901>:
0x458b0975
(gdb) set {char}0x00004c9d=0x34
(gdb) x/x 0x00004c9c
0x4c9c <-[SSExcesAppController awakeFromNib]+901>:
0x458b3475
```

Here we set the first byte at `0x00004c9c`. By simply counting in hexadecimal, we know that the next byte goes at address `0x00004c9d`, and set it as such. Let's disassemble again to check if the change was done right,

```
(gdb) disas
0x00004c9c <-[SSExcesAppController awakeFromNib]+901>:
jne 0x4cd2 <-[SSExcesAppController awakeFromNib]+955>
```

Whoops, we made a mistake and changed the destination of the jump from `+912` to `+955`. We realize that the first byte (`74`) of the byte code stands for the `je/jne` and the second byte is the offset, or how many bytes to jump by. We should only have changed `74` to `75`, and not `09` to `34`. Let's fix our mistake.

```
(gdb) set {char}0x00004c9c=0x75 (gdb) set
{char}0x00004c9d=0x09
```

And check again,

```
0x00004c9c <-[SSExcesAppController awakeFromNib]+901>:
jne 0x4ca7 <-[SSExcesAppController awakeFromNib]+912>
```

Hooray! This looks good! Let's execute the app to admire our crack.

```
(gdb) continue
```

Woot! Victory! We're in, and the application thinks we're a legitimate customer. Time to get wasted and party! (I recommend Vessel nightclub in downtown San Francisco.) Well, not quite. We still need to make our change permanent. As it currently stands, everything will be erased as soon as we quit gdb. We need to edit the code on disk — the actual binary file. Let's find a chunk of our edited binary big enough that it likely won't be repeated in the whole binary.

```
(gdb) x/8x 0x00004c9c
0x4c9c <-[SSExcesAppController awakeFromNib]+901>:
0x458b0975  0x2c40c608  0x8b7deb01  0xa4a10855
0x4cac <-[SSExcesAppController awakeFromNib]+917>:
0x89000141  0x89082454  0x89042444  0x26e82414
```

That's the memory representation of the code, a whole 8 blocks of four-bytes starting at `0x00004c9c`. Taking endianness into account, we must reverse them and we get the following:

```
0x75098b45  0x08c6402c  0x01eb7d8b  0x5508a1a4
0x41010089  0x54240889  0x44240489  0x1424e826
```

The very first byte of the series is the `74` that we switched into `75`. By changing it back, we can deduce the original binary code to be:

```
0x74098b45  0x08c6402c  0x01eb7d8b  0x5508a1a4
0x41010089  0x54240889  0x44240489  0x1424e826
```

Let's open the binary in a hex editor. I used `vim`, but feel free to use any hex editor at this point. HexFiend has a great GUI.

```
(gdb) quit
$ vim Exces
```

This loads up the binary as ascii text, which is of little help. Convert it to hex thusly,

```
:%!xxd
```

`vim` formats hex like this,

```
0000000: cafe babe 0000 0002 0000 0012 0000 0000
................
```

The first part, before the colon, is the address of block. Following it are sixteen16 bytes, broken off in two-byte segments. Incidentally, every Mach-O binary starts with the hex bytes `cafebabe`. Drunk Kernel programmers probably thought it'd be funny. Now that we have our beautiful hex code loaded up, let's search for the first two bytes of our code to replace:

```
/7409
```

Shit. Too many results to make sense of. Let's add another two bytes. Search for "7409 8b45" instead and boom, only one result,

```
001fc90: 0089 4424 04e8 4b24 0100 85c0 7409 8b45
..D$..K$....t..E
```

Edit it to the following,

```
001fc90: 0089 4424 04e8 4b24 0100 85c0 7509 8b45
..D$..K$....t..E
```

Convert it back to binary form, then save and quit,

```
:%!xxd -r :wq
```

And… we're done! To check our work, launch the application in `gdb`, break to `[SSExcesAppController awakeFromNib]` and disassemble.

```
$ gdb Exces
(gdb) break [SSExcesAppController awakeFromNib]
Breakpoint 1 at 0x4c90
(gdb) r
(gdb) disas
```

Admire our work,

```
0x00004c9c <-[SSExcesAppController awakeFromNib]+901>:
jne 0x4ca7 <-[SSExcesAppController awakeFromNib]+912>
```

Quit `gdb` and relaunch the application from the Finder, and bask in your *leet* glory.

How you can stop me

Objective-C makes it really easy to mess with an application's internals. Try to program the licensing mechanism for your application in pure C, that will already make it harder for me to find my way around your binary. Also, read this older article of mine [hn.my/binarycrack] on three easy tips — stripping debug symbols, using PT_DENY_ATTACH, and doing a checksum of your binary — you can implement to make it a whole lot harder for your application to be cracked.

A truly skilled hacker will always find his way around your protection, but implementing a bare minimum of security will weed out 99% of amateurs. I am not a skilled hacker — yet with some very basic knowledge I tore this apart in no time. Implementing the various easy tips above takes very little time, yet would have made it enough of a pain for me that I would have given up. ■

From Lausanne, Switzerland, Kenneth Ballenegger moved to the Bay Area where he works as an iPhone developer. When not busy building the next big thing, he studies graphic design during the day, while roaming San Francisco's trendy spots at night. Kenneth muses about the world of design, software and life on his blog, *kswizz.com*.

Reprinted with permission of the original author. First appeared in *hn.my/crackapp*.

Write Code Like You Just Learned How To Program

By JAMES HAGUE

I'm reading *Do More Faster*, which is more than a bit of an advertisement for the TechStars startup incubator, but it's a good read nonetheless. What struck me is that several of the people who went through the program successfully enough to at least get initial funding didn't know how to program. They learned it so they could implement their startup ideas.

Think about that. It's like having a song idea and learning to play an instrument so you can make it real. I suspect that the learning process in this case would horrify most professional musicians, but that horror doesn't necessarily mean that it's a bad idea, or that the end result wouldn't be successful. After all, look at how many bands find success without the benefits of a degree in music theory.

I already knew how to program when I took an "Introduction to BASIC" class in high school. One project was to make a visual demo using the sixteen-color, low-resolution mode of the Apple II. I quickly put together something algorithmic, looping across the screen co-ordinates and drawing lines and changing colors. It took me about half an hour to write and tweak, and I was done.

I seriously underestimated what people would create.

One guy presented this amazing demo full of animation and shaded images. I'm talking crazy stuff, like a skull that dripped blood from its eye into a rising pool at the bottom of the screen. And that was just one segment of his project. I was stunned. Clearly I wasn't the hotshot programmer I thought was.

Eventually, I saw the BASIC listing for his program. It was hundreds and hundreds of lines of statements to change colors and draw points and lines. There were no loops or variables. To animate the blood he drew a red pixel, waited, then drew another red pixel below it. All the coordinates were hard-coded. How did he keep track of where to draw stuff? He had a piece of graph paper that he updated as he went.

My prior experience hurt me in this case. I was thinking about the program, and how I could write something that was concise and clean. The guy who wrote the skull demo wasn't worried about any of that. He didn't care about what the code looked like, or how maintainable it was. He just wanted a way to present his vision.

There's a lesson there that's easy to forget — or ignore. It's extremely difficult to be simultaneously concerned with the userexperience of whatever it is that you're building and the architecture of the program that delivers that experience. Maybe impossible. I think the only way to pull it off is to simply not care about the latter. Write comically straightforward code, as if you just learned to program, and go out of your way to avoid wearing any kind of software engineering hat — unless what you really want to be is a software engineer, and not the designer of an experience. ■

James Hague is a recovering programmer who now works full time as a game designer, most recently acting as Design Director for Red Faction: Guerrilla. He's running his own indie game studio and is a published photographer.

Reprinted with permission of the original author. First appeared in *hn.my/codelearn*.

How Do Emulators Work And How Are They Written

By CODY BROCIOUS

Emulation is a multifaceted area. Here are the basic ideas and functional components. Many of the things I'm going to describe will require knowledge of the inner workings of processors — assembly knowledge is necessary.

Basic Idea

Emulation works by handling the behavior of the processor and the individual components. You build each individual piece of the system and then connect the pieces much like wires do in hardware.

Processor Emulation

There are three ways of handling processor emulation:

- Interpretation
- Dynamic recompilation
- Static recompilation

With all of these paths, you have the same overall goal: execute a piece of code to modify processor state and interact with "hardware". Processor state is a conglomeration of the processor registers, interrupt handlers, etc, for a given processor target. For the 6502, you'd have a number of 8-bit integers representing registers: `A`, `X`, `Y`, `P`, and `S`; you'd also have a 16-bit `PC` register.

With interpretation, you start at the `IP` (instruction pointer — also called `PC`, program counter) and read the instruction from memory. Your code parses this instruction and uses this information to alter processor state as specified by your processor. The core problem with interpretation is that it's very slow; each time you handle a given instruction, you have to decode it and perform the requisite operation.

With dynamic recompilation, you iterate over the code much like interpretation, but instead of just executing opcodes, you build up a list of operations. Once you reach a branch instruction, you compile this list of operations to machine code for your host platform, then you cache this compiled code and execute it. Then, when you hit a given instruction group again, you only have to execute the code from the cache. (By the way, most people don't actually make a list of instructions but compile them to machine code on the fly — this makes it more difficult to optimize)

With static recompilation, you do the same as in dynamic recompilation, but you follow branches. You end up building a chunk of code that represents all of the code in the program, which can then be executed

with no further interference. This would be a great mechanism if it weren't for the following problems:

- Code that isn't in the program to begin with (e.g., compressed, encrypted, generated/modified at runtime, etc) won't be recompiled, so it won't run.
- It's been proven that finding all the code in a given binary is equivalent to the "Halting problem".

These combine to make static recompilation completely infeasible in 99% of cases. For more information, Michael Steil has done some great research in static recompilation — the best I've seen.

The other side to processor emulation is the way in which you interact with hardware. This really has two sides:

- Processor timing
- Interrupt handling

Processor Timing

Certain platforms — especially older consoles like the NES, SNES, etc — require your emulator to have strict timing to be completely compatible. With the NES, you have the PPU (pixel processing unit) which requires that the CPU puts pixels into its memory at precise moments. If you use interpretation, you can easily count cycles and emulate proper timing; with dynamic/static recompilation, things are a lot more complex.

Interrupt Handling

Interrupts are the primary mechanism that the CPU communicates with hardware. Generally, your hardware components will tell the CPU what interrupts it cares about. This is pretty straightforward — when your code throws a given interrupt, you look at the interrupt handler table and call the proper callback.

Hardware Emulation

There are two sides to emulating a given hardware device:

- Emulating the functionality of the device.
- Emulating the actual device interfaces.

Take the case of a hard drive. The functionality is emulated by creating the backing storage, read/write/format routines, etc. This part is generally very straightforward.

The actual interface of the device is a bit more complex. This is generally some combination of memory mapped registers (e.g., parts of memory that the device watches for changes to do signaling) and interrupts. For a hard drive, you may have a memory mapped area where you place read commands, writes, etc, then read this data back. ■

For a complete list of resources and addendum, see the original post at *hn.my/emulators*.

Cody Brocious is a senior security consultant with Matasano Security and founder of projects such as PyMusique, Emokit, and the Renraku OS. His work has been featured in Forbes Magazine, EDN, CNET's *News.com*, and on his mother's refrigerator.

Reprinted with permission of the original author. First appeared in *hn.my/emulators*.

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