CSM CompSci Field Session 2012: 'dd-able' Live USB Image Generation for Oracle Solaris
Final Report

Authors:
Andrew Grossnickle
Earl Colin Pilloud
Zachary Harrison Stigall

June 19, 2012

1 Introduction

1.1 Client Description

Oracle is an international software giant, specializing in enterprise software solutions with an eye towards data management. One of these solutions is the Oracle Solaris operating system, a state-of-the-art, cutting-edge enterprise OS built from the ground up for cloud computing. Solaris provides a fertile platform for building and running cloud-based applications.

For users who are interested in trying out Solaris, Oracle provides a live DVD/USB image for download, allowing them to take the OS for a test drive before installing it on their machine. However, in order to copy the downloaded image to a USB drive, users must use an Oracle utility named ‘usbcopy’, which is native to Solaris. Users who are looking to try out Solaris may find themselves stonewalled at this point, caught in a catch-22: in order to install Solaris, they need Solaris.

1.2 Product Vision

In order to connect and appeal to more potential users of the Solaris operating system, a more robust process for creating a live USB media must be created. This process will grant users of most operating systems the ability to install and test Solaris.

Our challenge was to design and implement a solution that directly generates a USB image that is dd’able, providing a simple alternative to using the ‘usbcopy’ script. dd is a utility common on Unix based operating systems that does a bytewise copy from one source to another. Running dd with a USB image generated by our solution as an input and a USB drive as an output should result in a bootable drive, enabling users of most operating systems to get up and running with Solaris in no time. This simple change makes trying Solaris more convenient for users, hopefully resulting in an increase in the overall number of converted users.

2 Requirements

2.1 Functional

1. The solution must create an image that is dd’able to a USB drive.
2. Images created using the solution must utilize UEFI and GRUB2 to boot.
3. The solution must exist within and cooperate with the existing codebase.
4. The solution should be able to be run standalone, in order to convert an existing non-dd’able image into one that is dd’able.
5. Backwards compatibility should be maintained; if our solution is not run in the course of producing a new image, the behavior should continue as it is now.
6. The solution must work for all past versions of Solaris, in addition to all future versions.
7. The solution should contain extensive and configurable logging.
8. The solution should fail fast and gracefully, with appropriate error checking and handling.

2.2 Non-functional

1. The solution must be written in Python.
2. The solution must conform to the Python 2.6 standard.

3. Mercurial version control must be used.

3 System Architecture

Our utility will need to work within the codebase currently used by the Solaris installation. This means that the options of how our utility can interface with this codebase are, to some extent, predetermined. Currently, a Solaris installation image is created by an Oracle utility called ‘distro_const’, short for distribution constructor. ‘distro_const’ contains checkpoints, essentially scripts that execute sequentially, each contributing towards the full, final image. All of the tools used by ‘distro_const’ are located in a developer directory named ‘slim_source’. This directory also contains the ‘usbcopy’ script that’s currently used to move USB images. The general flow of the USB image creation and installation to a drive is detailed below in Figure 1.

Figure 1

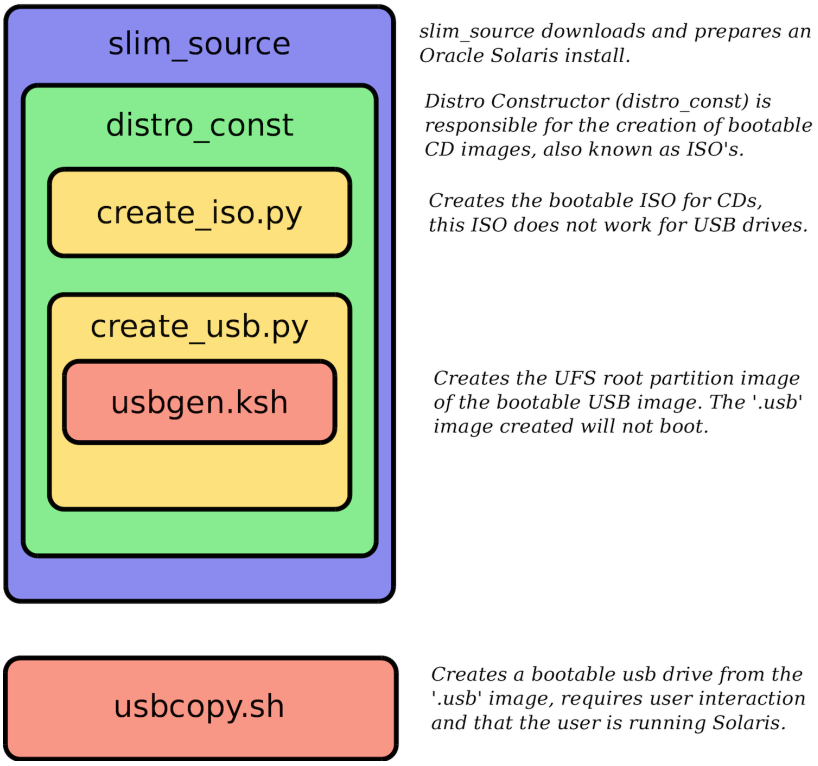

Figure 1 - System architecture for image creation/installation process.

After investigating the behavior of the ‘usbcopy’ script, we identified why the image was not bootable. In order to boot, a drive needs a master boot record (MBR) and information about the partitions, along with a boot manager. ‘usbcopy’ contains logic that prepends MBR and GRUB (either legacy or version 2.0, depending on what’s contained in the image) files to the .ISO file. When the computer starts up, it sees the boot record on the drive and correctly identifies the drive as bootable. Without the MBR and GRUB files, the USB images produced by ‘distro_const’ were not bootable; ‘usbcopy’ contains the logic to place those boot files in the correct spot, so our challenge was to port that behavior into our utility.

Our solution involves modifying ‘distro_const’ to call a new script at an appropriate checkpoint. This script would accept an image as an argument, modify the image to be dd’able, then return

the image back to the ‘distro_const’. As we discovered, this involved configuring the USB image to contain a GRUB2 config, some MBR information, and setting up the partition tables; what was nice about this solution was that most of the work with the booting configuration was already done inside the ‘usbcopy’ utility, allowing us to use that as a guide.

Figure 2

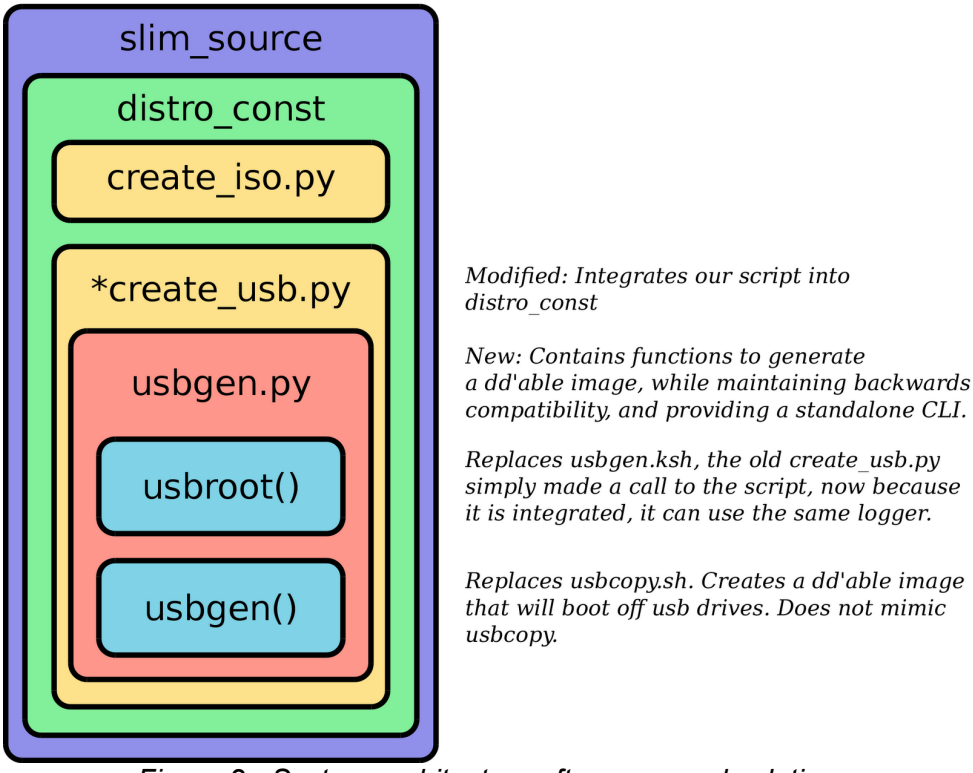

Figure 2 - System architecture after proposed solution

We then identified the correct place within ‘distro_const’ for our utility to execute: one of the checkpoints inside ‘distro_const’ called ‘create_usb.py’. This is the checkpoint that converts an .ISO image into one suitable for use on a USB drive. The checkpoint in the current system is essentially just a Python wrapper for a shell script called ‘usbgen’ that does the heavy lifting, converting the file system used in the .ISO image into a Unix File System (UFS) and copying the files over to the new USB image. We changed the checkpoint to refer to our utility instead, incorporating the behavior of the ‘usbgen’ script into our utility. The changes made to the overall USB image creation process are detailed in Figure 2, above.

4 Technical Design

Our utility contains two core methods: one to duplicate the behavior of ‘usbgen’, and one to prepend the MBR and GRUB files to the image (the behavior contained in ‘usbcopy’). The first function, ‘usbroot’, creates and mounts a loopback device used to build the final image. It then formats the new image with the UFS filesystem, then copies over the contents of the .ISO image using the ‘cpio’ utility, which preserves file metadata. The second function, ‘usbgen’, takes the image produced by ‘usbroot’ and grabs the UEFI and GRUB files from within. Information and statistics about those files and the USB image itself are used to calculate a few lengths

necessary for modifying the partition table and boot record, then the three files are combined using a simple ‘cat’ command. The partition table is then modified using the sizes we calculated earlier so the computer knows where the root partition begins and ends. Finally, the first 440 bytes need to be copied over again, since the command used to update the partition table unnecessarily modified these very important bytes.

With the script completed, we began working on implementing logic to allow it to function as a command line utility, satisfying the standalone requirement. We used Python’s ‘optparse’ library to allow for easy option and argument parsing, used when the script is executed from the command line. Exploration of the library enabled us to do some cool things. A help and usage message is displayed when the user passes in bad input or asks the program for help, with the message detailing what exactly the user did wrong and how to correct it.

When run from the command line, our utility offers two options. One, ‘--legacy’, fires only the first method, giving the user the old behavior provided by the original ‘usbgen’ script. The other option is ‘--current’, allowing the user to convert a non-dd’able USB image into one that has that functionality.

One of our favorite features, especially when it came to testing, was a smart cleanup function. Our two methods do a lot of device manipulation, mainly mounting and dismounting. We also make use of a couple temporary directories, so we had to make sure we put everything back in order after our script finished executing, either successfully or in the case of failure. We wrapped our two methods in a class and added a series of boolean flag instance variables, used to indicate how far along the script managed to get before exiting. Those flags allowed our cleanup function to do only what it needed to, which was for the best - we didn’t want the script to delete files unnecessarily or attempt to unmount drives that never existed.

To guarantee that the cleanup function got called we utilized a very nice Python feature: the ‘with’ statement. The with statement creates an object and upon any sort of exit, error or not, runs a special function defined in the class as __exit__(). This made error handling very easy, allowing us to avoid verbose and complicated try-catch-finally statements.

5 Design & Implementation Decisions

In developing the design for our solution, we considered a pair of alternative designs. These alternatives included modifying the image produced by the distribution constructor and creating an image that contains a GRUB2 install that boots into the ISO created by the distribution constructor.

5.1 Modify the distro_const to create a dd’able image

This consideration ended up looking remarkably similar to the solution we ended up with, with one key difference. We would edit a checkpoint further up the chain in ‘distro_const’ to correctly set up the boot record and partition tables for all future images produced in ‘distro_const’, thus making them all bootable. The checkpoint we had in mind was the ‘create_iso.py’ - if the image that checkpoint produced was already dd’able, then the .ISO image that ‘create_usb.py’ relied on would also be dd’able, allowing the USB checkpoint to skip that step.

We chose not to move forward with this solution because we felt it had a large impact on ‘distro_const’; our focus was on producing dd’able USB images. While the solution would

have worked, we didn't want to touch images that wouldn't end up as USB images. In short, the 'create_iso.py' happens too soon in the image creation process for our change to have only the desired impact, with no unintended side effects.

5.2 Create GRUB2 install that boots into an .ISO from distro_const

This consideration involved creating a GRUB2 install that would boot into an .ISO generated by the 'distro_const'. This process would not modify the existing .ISO in any way; updating the .ISO would be a simple file swap on the USB drive. A dd'able image would be created containing the given .ISO and GRUB2 setup. In short, a USB image would be generated by combining a GRUB2 install with any .ISO from the distro_const, which would in turn be dd'able by most operating systems.

This solution was simple and effective, but it represented a radical departure from the current behavior of 'usbcopy'. We did not pursue this solution because we felt like we should mimic the logic that was already in place - logic that was known to work correctly and was trusted by Oracle. This process felt too "hackish" by comparison.

5.3 Other Considerations

One design choice in particular was made for us by the client - the language used for our utility. Coding in bash would have been a simple affair, because both of the scripts whose behavior we were mimicking were written in shell, and combining them would have been trivial. The Solaris installation team has made an effort to port their codebase into Python, so that was chosen as the required language.

6 Results

6.1 Test Results

Our test process involved running the utility with several different images, then dd'ing them onto a USB drive and plugging that into a computer, which tested bootability.

We tested our utility by taking several live images from the Solaris downloads page. We obtained several .ISO files for both production and cutting edge versions of Solaris and ran them all the way through our utility, i.e. running both methods on the image. This resulted in a dd'able USB image in every case. We also took live USB images from the downloads site and ran our utility on them, using the '--current' option from the command line interface. In each case, the resultant image was dd'able, indicating our utility does exactly what's intended.

Due to technical difficulties, our utility has not been tested in context with 'distro_const'. We were provided a Solaris platform for use as a central code repository and also for use as an integration testing machine. This Solaris box did not have the hardware capability to build and run the 'distro_const'. Furthermore, we would need access to Oracle's internal network in order to download the tools needed to build and run the 'distro_const' on another machine. With all of our last minute activities, we did not get a chance to access those tools, and as a result, our integration testing is, incomplete. However, we are certain that our utility will be able to integrate with minimal modifications. Our utility is written in such a way that its core functionality is the same for both the standalone version and the version that is integrated. Since we are

positive this functionality works, integration should take minimal effort.

6.2 Future Work

We provided a base level of logging in our utility; when called from the checkpoint, the checkpoint passes in its logger as an argument for the utility. The installation team may wish to expand upon the way logging is handled, and possibly the various logging levels. We defaulted every log statement to use the logging level of debug, with the exception of errors, which register at the level of error.

The utility should be installed/integrated using the directions provided in appendix 7.1.

7 Appendices

7.1 Installation & Integration

In order to properly integrate our utility into the distribution creation process, the following changes should be made:

1. Our utility, ‘usbgen.py’, should be placed in the directory that makes the most sense to the install team - slim_source/usr/src/cmd/install-tools makes the most sense to us, because that’s where the current usbgen script lives.

2. ‘create_usb.py’ should then import our utility and call it, most likely by creating an instance of the class, setting the instance variables necessary, and then calling both methods. Adjustments to the surrounding variables may need to be made.

3. The ‘usbgen’ shell script has been incorporated into our utility, so it can be deleted at the earliest convenience. ‘usbcopy’ will still be necessary to move images created with our utility’s ‘--legacy’ option, so it can stay until users are used to dd’ing their USB images instead of using ‘usbcopy’.

7.2 Coding Conventions

We learned about Oracle’s coding style by reading through other Python scripts in slim_source, and by reading Python’s own coding conventions in PEP8. We tried our best to adhere to those standards. Our lines are all eighty characters or less, and we used 4 space soft-tabs for indentation.

We discussed logging above, but we’ll summarize here. We handle logging by passing the Solaris logger from ‘create_usb.py’ as an argument to our utility. When run standalone at the command line, our utility uses the default root logger, feeding all logging information to standard out. Members of the Solaris team may wish to fine-tune the logging to better match the rest of the codebase.

7.3 Scrum Reports

Here’s what our project backlog looked like when finished. Actual results for the final report and presentation are still in progress.

Overall Project: 'dd-able' Live USB Image Generation

		1 point = 4 hours		
ID #	Story / Feature / Request	Estimate (points)	Actual (Points)	Worked on by
1	Dev environment configuration	4	5	AG, EP, ZS
2	Python/Shell scripting research	4	3	AG, EP, ZS
3	Image/Booting research	8	10	AG, EP
4	Design specification creation	2	2.5	EP
5	Development of prototype script	3	2	AG, ZS
6	Creation of Python usbgen script	4	4	EP, ZS
7	Incorporation of old usbgen script	4	7	EP,
8	Argument Handling	2	4	EP
9	Logging	3	4	AG, EP, ZS
10	Error Handling	4	2	EP, ZS
11	Documentation	2	3	EP, ZS
12	Integration into current codebase	3	6	AG
13	Final Report	4		AG. EP, ZS
14	Final Presentation	4		AG, EP, ZS